TURING 图灵程序设计丛书

MANNING

- 敏捷开发新经典文献
- 亚马逊网站全五星评价
- 成功开发软件的必备秘籍

实例化需求

团队如何交付正确的软件

[塞尔维亚] Gojko Adzic 著　　张昌贵　张博超　石永超　译

Specification by Example

How Successful Teams Deliver the Right Software

人民邮电出版社
POSTS & TELECOM PRESS

图灵程序设计丛书

实例化需求：团队如何交付正确的软件

Specification by Example: How Successful Teams Deliver the Right Software

[塞尔维亚] Gojko Adzic 著 张昌贵 张博超 石永超 译

人民邮电出版社

北京

图书在版编目**(CIP)**数据

实例化需求：团队如何交付正确的软件/(塞尔)阿契克(Adzic,G.)著：张昌贵，张博超，石永超译.--北京：人民邮电出版社，2012.9

(图灵程序设计丛书)

书名原文：Specification by Example: How Successful Teams Deliver the Right Software

ISBN 978-7-115-29026-7

Ⅰ.①实… Ⅱ.①阿… ②张… ③张… ④石… Ⅲ.①软件开发—电子计算机工业—工业企业管理—组织管理 Ⅳ.①F407.676.17

中国版本图书馆CIP数据核字(2012)第168201号

内容提要

本书是在世界各地调查了多个团队软件交付过程后的经验总结。书中介绍了这些团队如何在很短的周期内说明需求、开发软件，并交付正确的、无缺陷的产品；为团队在实施实例化需求说明时使用的模式、想法和工件创建了一致的语言；展示了案例中的团队用来实现实例化需求说明原则的关键性实践；并在案例分析部分展示了一些团队实施实例化需求说明的历程。

本书适合与项目管理、开发、测试、交付有关的人员阅读。

图灵程序设计丛书

实例化需求：团队如何交付正确的软件

◆著　[塞尔维亚] Gojko Adzic

译　张昌贵 张博超 石永超

责任编辑 傅志红

执行编辑 李瑛

◆人民邮电出版社出版发行　北京市崇文区夕照寺街14号

邮编 100061　电子邮件 315@ptpress.com.cn

网址 http://www.ptpress.com.cn

北京印刷

◆开本：800×1000 1/16

印张：13

字数：307千字　2012年9月第1版

印数：1-4000册　2012年9月北京第1次印刷

著作权合同登记号 图字：01-2012-4260号

ISBN 978-7-115-29026-7

定价：49.00元

读者服务热线：**(010)51095186**转**604** 印装质量热线：**(010)67129223**

版权声明

前言

你手里正拿着或正在屏幕上翻看的这本书，是一系列研究的成果。我们调查了世界各地的多个团队如何在很短的周期内说明需求、开发软件，并交付正确的、无缺陷的产品。本书呈现的是集体智慧，从公共网站到内部支持系统，涉及大大小小约50个项目。这些项目包含了各种各样的团队，有在一个办公室里办公的小团队，也有跨越大洲的集团公司，他们使用了众多过程，包括极限编程(XP)、Scrum、看板(Kanban)以及一些类似的方法（通常附带有敏捷或精益的字眼）。这些项目有个共同点——项目需求说明和测试能够良好配合，项目组从中获益良多。

实例化需求说明

如何处理需求说明与测试，不同的团队使用不同的名称，但它们都有一套共同的核心原则与思想，而我认为它们在本质上是一致的。很多团队使用以下这些名称来命名这类实践：

敏捷验收测试

验收测试驱动开发

实例驱动开发

故事测试

行为驱动开发

实例化需求说明

相同的做法却有如此多的名字，这事实上也反映了当前在这一领域内有着大量的创新。同时它还反映了一个事实，本书描述的这些做法，影响了团队的需求描述、开发和测试等方面。为保持一致，必须选择一个名字。本书将采用实例化需求说明(Specification by Example)这个名称。至于为何选它，稍后的“谈谈术语”一节将详细解释。

实践出真知

本书通过案例研究和访谈来呈现这个话题。之所以选择这种方式，是为了让读者能看到目前真的有团队正在这么做，并且从中获益良多。实例化需求不是一门神秘艺术，虽然有些主流媒体会使人这么觉得。

书中的一切几乎都是来自于现实世界、真实的团队以及切实的经验。有一小部分实践是作为建议提出的，并没有真实案例研究的支持。我认为这些思想对将来会很重要，也正是因为如此，我才明确地提出它们。

我很确定，我所主导的研究以及我得出的结论，虽然促成了本书的编写，但由于这并不是一项严肃的科学研究，将会被那些号称敏捷开发

不可用、业界应该回到“真正的软件工程”[①]的怀疑论者所排斥。那也没关系。相比一项严肃的科学研究所需的资源，编写本书时我接触到的资源是十分有限的。但即使我拥有那些资源，我也不是一个科学家，而且我也不用伪装我自己。我是一名实践者。

注释：①关于工程学的严谨有助于软件开发的错觉（如同二流的物理学分支），可参见http://www.semat.org。想更多了解对此错觉的反击，请参考Glenn Vanderburg的演讲“软件工程没用！”(http://confreaks.net/videos/282-lsrc2010-real-software-engineering)。

本书读者对象

如果你像我一样，是一名实践者，并且靠软件谋生，那么这本书可以提供很多帮助。有些团队已经尝试去实施敏捷过程，并且碰到了低质量、返工以及未达客户预期等问题，本书主要就是写给这些团队的。（没错，这些都是问题。简单地迭代只是权宜之计，并非解决方案。）实例化需求说明、敏捷验收测试、行为驱动开发，以及其他不同叫法所指的这个实践，都能解决这些问题。无论你是测试人员、开发人员、业务分析师，还是产品负责人，这本书都可以帮助你开始实施这些实践，并学习如何用它们在团队中做出更多贡献。

几年前，我在大会上碰到的大多数人都没听说过这些思想。而现在，我碰到的大部分人都或多或少知道这些实践，但是很多人都未能妥善落实。在实施敏捷开发的过程中，团队碰到的问题通常都很少有文字记载，所以每一个受挫的团队都认为，自己遇到的问题比较特殊，而这些理念无法在他们的“现实世界”里发挥作用。只需听他们述说短短五分钟，我就能猜中三四个他们碰到的最大问题，这让他们觉得惊讶。而当他们得知其他团队也有同样的问题时，他们更是完全惊呆了。

如果你也在这样的团队当中，那么本书为你做的第一件事情，将是告诉你“你并不孤单”。本书中我所采访的那些团队并不完美——他们也曾遇到数不清的问题。但他们在碰壁之后，并没有放弃，而是决定围绕这些问题继续努力并解决问题。了解这一点通常能鼓舞人们换一种眼光去看待自己的问题。我希望你在读罢本书后也有同样的感受。

如果你正在实施实例化需求说明，本书将就如何解决你当前的问题提供非常有用的建议，同时也能让你了解未来会发生什么事情。我希望你可以从别人的错误中汲取经验，并且完全避免发生同样的问题。

本书也写给有经验的实践者，以及那些在实施实例化需求说明的过程中相对成功的人们。在采访开始之初，我本以为这些事情都已胸有成竹，只是在求证而已。结果我发现，人们在实施过程中居然有如此之多

的想法，这是我始料未及的。我从这些案例中学到了很多，希望你也如此。这里所描述的实践和想法，应该会激发你的灵感，促使你对自己的问题尝试变通方案，或者让你在见过类似的故事之后，意识到可以如何改善团队的过程。

本书内容

在第一部分，我会介绍实例化需求说明。我不会说服你为什么应该遵循本书描述的原则，而是向你展示——用实例化需求说明的方式——团队从这个过程中获益的例子。如果你在考虑购买这本书，请浏览一下第1章，看看有哪些好处可以带到你的项目中。第2章介绍了关键的过程模式和实例化需求说明的关键工件(artifact)。在第3章，我会更详细地解释活文档的概念。第4章会展示一些改变过程和团队文化的最常见的切入点，也会就开始过程实施时需要注意的地方给出一些建议。

本书写作的一个目的是为团队在实施实例化需求说明时使用的这些模式、想法和工件创建一致的语言。整个实践在业界有许多名称，里面各种要素的名称更是多出一倍。不同的人分别将同一个东西叫作功能文档、故事测试、BDD文档、验收测试等。正因为如此，在第2章中我会对所　有的关键要素介绍一些我感觉不错的名字。即使你非常有经验了，我还是建议阅读这1章，以确保我们对本书中的关键名字、用词和模式的理解是一样的。

在第二部分，我会展示案例中的团队用来实现实例化需求说明原则的关键性实践。不同环境中的团队会做非常不同的事，有时甚至为了达到相同效果采取相反或冲突的措施。除了实践外，我还记录了团队贯彻基本原则的环境。第二部分差不多按照过程区域分成7章。

在软件领域没有最佳实践，但是确实有一些好的想法我们可以尝试在不同的环境中去使用。在第二部分中，有些小节标题旁边会有拇指向上或向下的图标，代表的是调查中一些团队觉得有用的做法，或者是他们经常遇到的问题。请根据建议做适当的尝试或回避，但不要完全照搬套用。箭头图标出现的地方代表的是各种做法的精髓。

软件开发不是静态的——团队和环境都在改变，过程也必须随着改变。在第三部分中，案例分析展示了一些团队的实施历程。我记录了他们的过程、约束条件和环境，并分析了这些过程是如何演化的。这些故事有助于你迈开第一步或让你更进一步，并发现新的想法与做事方式。

本书的最后一章，我总结了我从促成本书的案例分析中学到的关键要素。

更上一层楼

在传统的学习模型守破离(Shu-ha-ri)[①]中，本书处于破的层次。破是说要打破陈旧的规则，并证明成功的模型有很多。在我的Bridging the

communication Gap一书中，我展示了我的模型及经验。本书中，我尽量不带进以前的背景。只有当我觉得有重要观点需要证明，并且本书中提到的其他任何团队都没有类似情况的时候，我才会展示那些我自己参与过的项目。从这个意义上讲，本书在Bridging the Communication Gap的基础上更进了一步。

注释：①“守破离”是来自于“合气道”（日本的一种自卫拳术）招式的学习模型。它包含三个层次。第一层“守”，学员必须严格学习一种招式。第二层“破”，学员知道除了他所学的招式外还有很多招式。第三层“离”，学员脱离招式的束缚。

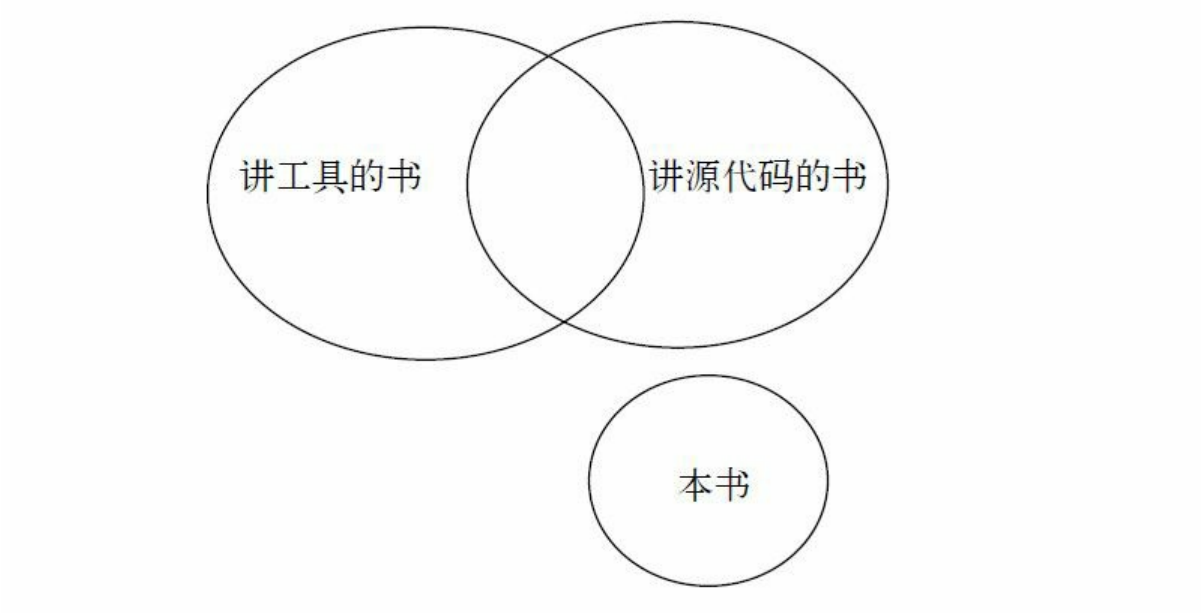
我会在第2章简单介绍一些基本的原则。即使你以前从未听说过这些想法，第2章的简介也应该可以给你足够的信息去理解本书的其余部分，但我不会过多地深入基础的内容。有关实例化需求说明的基础内容在Bridging the Communication Gap一书中有详细的描述，我不想在本书中重复。

如果你想更详尽地重温那些基础内容，请访问http://specificationbyexample.com，登记你购买了本书，就可免费获得Bridging the Communication Gap的PDF版本。

我想今后我不会就这一主题续写“离”这个层次的书籍——因为该层次是超越书籍的。另一方面，我相信本书可以帮助你到达“离”这一层次。一旦你开始觉得选择什么工具已经无关紧要，那么你就已经达到了这个层次。

本书没有源代码，也不介绍任何工具

本书没有源代码，也没有特定工具的使用说明。我觉得必须事先说明这一点，因为在出版过程中，我就曾多次向别人解释这一点（典型的问题有“什么意思？一本没有源代码的软件开发书？这怎么可能！”）。



实例化需求说明的原则和实践主要影响软件交付团队中的人员沟通，以及他们如何同使用者和项目干系人进行协作。我确信许多工具供应商会试图卖给你一套技术方案。如果有工具可以立即消除遇到的问题，许多经理会乐于为此买单。不幸的是，他们遇到的主要是人的问题，而不是技术问题。

比尔·盖茨说过："在企业中应用任何一项技术时，首要的法则是，在有效率的系统中导入自动化，将使效率倍增。第二条法则是，在缺乏效率的系统中导入自动化，会使效率更低下。"很多团队在使用实例化需求说明的时候失败了，他们使用自动化工具反而导致他们的过程更加低效。我不想把注意力放在特定的工具上，相反，我想侧重分析团队努力实现这些想法的真实原因。一旦你们能正确地沟通和协作，你们就可以选择适合的工具去使用。在阅读本书后，如果你想知道更多支持实例化需求说明的工具，请访问http://specificationbyexample.com并查看资源部分。

谈谈术语

如果这是你首次听说实例化需求说明、验收测试驱动开发、敏捷验收测试、行为驱动开发，或者人们为这类做法所起的任何其他名字，你应该庆幸自己没有被这些误导性的名字所困扰。你应该放轻松些，而且你可以跳过这个部分。如果你已经接触过那些做法，我在本书中使用的名字可能会让你感到惊讶。请接着读下去，这样你就能理解为什么我使用这些名字，并且你也应该开始使用它们。

在我编写本书的时候，我也遇到了实践者们在编写他们的自动化需求说明时经常遇到的问题。术语应该要一致，这样才能易于理解，当把

内容编写成文时很有必要明白这一点。本书是一系列访问的产物，很多我交谈过的人使用不同的名字来指称同一件事情，这样的话，要想保持所讲故事的一致就是相当困难的。

我意识到，实例化需求说明的实践者，包括我自己，通常会因为使用技术术语，导致我们自己以及其他尝试实施这些实践的人都很迷惑，这让我们感到内疚。因此我决定，编写本书的其中一个目标，就是要改变社区中使用的术语。让业务人员更多地参与进来是这些实践的一个主要目标，为此我们必须使用适当的名字去描述那些正确的做法，不要再让人们感到困惑。

当我们编写需求说明时，这个教训是显而易见的。我们都知道应该要保持术语的一致性，避免使用具有误导性的术语。但当我们谈论过程的时候，我们没有那么做。例如，当我们在实例化需求说明的环境中说持续集成的时候，我们并不是说要运行集成测试。因此，为什么要使用这个术语，然后不得不给其他人解释验收测试与集成测试的不同？在我开始使用需求说明工作坊(specification workshop)这个名字来代表有关验收测试的集体会议前，很难说服业务人员去参加。一个简单的名字变更就解决了这个问题。通过使用更好的名字，我们可以避免许多毫无意义的讨论，马上就让大家走上正确的道路。

为什么使用“实例化需求说明”这个名字

首先我想解释一下，为什么我选择实例化需求说明作为这些实践的总称，而没有使用敏捷验收测试、行为驱动开发或者验收测试驱动开发。

在2010年的伦敦领域驱动开发交流大会①上，Eric Evans跟别人争论，说敏捷作为一个术语已经失去了一切意义，因为现在什么都可以称为敏捷。很不幸的是，他是正确的。尽管有大量的著作讲如何正确地实施极限编程、Scrum以及其他不那么流行的敏捷过程，但我见过太多太多的团队，试图去实现冠以敏捷一词的过程，但那些过程又显而易见地违背了敏捷的精神。

注释：①http://skillsmatter.com/event/design-architecture/ddd-exchange-2010

为了避免这种关于敏捷是否可行（以及什么是敏捷）的无意义争论，在本书中，我尽量避免使用敏捷这一术语。只有当我提到的团队基于敏捷宣言概括的原则定义了良好的过程，并开始实施时，我才会使用它。由于不会经常提到敏捷这一术语，敏捷验收测试这个名称也就无从谈起了。

这里描述的实践没有形成一个成熟的软件开发方法论。它们可以补充其他方法论——无论是基于迭代还是基于工作流的——使需求说明和测试更加严谨，增强不同项目干系人和软件开发团队成员之间的沟通、减少不必要的返工，并让改变更加容易。因此我不想使用任何“驱动开发”之类的名字，尤其不会使用行为驱动开发(BDD)的字眼。不要认为这说明我反对BDD。恰恰相反，我喜欢BDD，而且我认为本书实际上主要在讲BDD的核心内容。但BDD同样有名字歧义的问题。

BDD到底代表了什么总是在变化。关于什么是BDD，什么不是BDD，Dan North是最具话语权的。在Agile Specifications，BDD，and Testing Exchange 2009①上，他说BDD是一种方法论。（事实上，他将其称为“第二代的、由外而内的、基于拉动的、多项目干系人、多尺度的、高度自动化的敏捷方法。”）为了避免在North所说的BDD和我理解中的BDD之间产生任何混淆和歧义，我不想使用这个名字。本书讲的是一组宝贵的实践，在很多方法论中你都可以使用，包括BDD（如果你能接受BDD是一种方法论的说法）。

注释：①http://skillsmatter.com/podcast/java-jee/how-to-sell-bdd-to-the-business

我也想尽量避免使用测试这个字眼。很不幸地，很多经理和业务人员认为测试是一种技术辅助活动，不是他们想参与的事情。毕竟，他们有专门的测试人员去处理这件事情。实例化需求说明要求项目干系人以及交付团队的成员（包括测试人员、开发人员、业务分析人员）积极地参与进去。只要我们在标题中不放入测试这样的词汇，那么故事测试、敏捷验收测试以及其他类似的名字自然就不会出现。

这让实例化需求说明成为了最有意义的名字，它的负面影响最小。

过程模式

实例化需求说明由一些过程模式(Process Pattern)组成，后者是更广义的软件开发生命周期的组成要素。本书中我用的名字是在英国敏捷测试用户组会议、敏捷联盟功能测试工具邮件组以及工作坊中经过一系列讨论得到的。其中有些名字已经用了一段时间，而另一些大多数读者还比较陌生。

业内流行用一个实践或工具的名字来描述过程的一部分。功能注入(Feature Injection)就是一个很好的例子——用它来描述从商业目标中获取项目范围这一过程，就很受欢迎。但是功能注入只是其中一种技术，还有其他方法可以达到同样的目的。为了讨论不同的团队在不同的环境中做什么，我们需要更宏观的概念来囊括所有这些实践。一个好的名字

能很好地说明预期的结果，并且清晰地指出这些实践的关键区别。

以功能注入与类似实践为例，其结果就是项目或里程碑的一个范畴。与其他定义范围的方法相比，关键区别在于，我们专注的是商业目标。因此我提出从目标中获取范围(deriving scope from goals)这一概念。

在实例化需求说明过程中，团队碰到的最大的一个问题是：谁应该在什么时候写些什么东西。所以我们需要一个好名字来明确地说明大家都应参与（需要在团队开始编码或测试前做），因为我们要使用验收测试作为开发的目标。测试先行(Test First)是个不错的技术名词，然而商业用户并不能理解，更何况它没有包含合作的意思。我建议我们关注通过协作制定需求说明(specifying collaboratively)，而非测试优先或写验收测试。

听起来很普通，只是把每个数字上的可能性考虑进自动化功能测试里。如果自动化了，我们为什么不这么做？但是如此复杂的测试作为沟通工具是无法使用的，在实例化需求说明里，我们是要用测试来沟通。因此不是编写功能测试，而是关注举例说明(illustrating using examples)，并且期望它能输出关键实例(key examples)这些实例表明我们只需要恰当地描述所需的环境就可以了[①]。

注释：①谢谢David Evans给的建议。

关键实例是原料，但是如果仅仅关注验收测试，为什么不干脆就用50列100行的复杂表格来作为实例，并且不带任何说明？反正是机器来测试。用了实例化需求说明，测试既是给机器看的，也是给人看的。我们需要说清楚的是，使用实例说明后，还有一个步骤，就是抽取属性和实例的最小集合来说明业务规则，并加上标题和描述等。我建议把这个步骤称为提炼需求说明(refining the specification)[②]。

注释：②谢谢Elisabeth Hendrickson建议的这个名字。

提炼的结果既是需求说明，同时也是开发的目标、检查验收的客观方法以及之后的功能回归测试。我不想叫它验收测试的原因是，它很难说明为什么文档要用领域语言来写、可读性要高，还要容易理解。我认为我们应该将提炼的结果称为带实例的需求说明(specification with examples)，从而揭示出一个事实，就是需求说明需要基于实例，但绝不是只包含原始数据。将这个工件称作需求说明可以明显地指出大家都需要关注它，而且它需要容易理解。除此之外，关于是否用这些检查来自动验收软件或自动拒绝不满足需要的代码，还有另外一种截然不同的观

点③。

注释：③http://www.developsense.com/blog/2010/08/acceptance-tests-lets-change-the-title-too

我不想再花时间和那些买了QTP的人争论，告诉他们QTP对验收测试完全没有用。只要我们谈论测试自动化，总有人推销测试人员已经用来做自动化的可怕玩意儿。敏捷验收测试和BDD工具与QTP之类的工具有很大的差别，它们处理完全不同的问题。不应将需求说明解释成一种自动化的技术。我们把在不歪曲任何信息的情况下将验证自动化称作自动化验证而不改变需求说明(automating validation without changing specifications)，而非验证自动化。不改变原有需求说明的自动化验证可以帮助我们避免可怕的脚本和在测试需求说明中直接使用技术类库。可执行的需求说明应该与在白板上看到的一样，而不应该转译成Seleninum命令。

在将需求说明验证自动化后，我们可以用它来验证系统。事实上，我们得到了可执行的需求说明(executable specification)。

我们要频繁地检查所有的需求说明，确保系统还在按所期望的运行，同样重要的是确保需求说明还能描述系统的行为。如果我们将这称作回归测试，那么很难向测试人员解释为什么他们不应该在之前既小又好而且明确的需求说明中再加50O万个测试用例。如果再提到持续集成，我们将很难解释为什么不总是端到端地运行这些测试，并检查整个系统。对于一些遗留系统，我们需要针对一个部署好的正在运行中的环境来运行验收测试。技术上的集成测试应在部署前运行。所以我们不谈回归测试或持续集成，我们谈频繁验证(validating frequently)。

实例化需求说明具有长期回报，前提是拥有一个对系统自身功能的引用，该引用具有与代码一样的价值，但是更易读懂。长期来说，这使得开发有效率得多，能促进与商业用户的合作，促成软件设计和商业模型的一致，并且使大家工作更简单。但是要做到这点，该引用必须有意义，必须有人维护，而且还必须内部保持一致，并与代码保持一致。我们不应该有大量的测试还在用数年前使用的术语。你很难让忙碌的团队回过头去更新测试，但是在重大改变之后回头去更新文档，这是大家所愿意看到的。所以我们不要关注那些含有数百个测试的文件夹，让我们把注意力放到演化活文档系统(evolving a living documentation system)上。这样就更容易解释，为什么很多事情必须不言而喻，为什么商业用户也需要使用这些，为什么需要良好地组织以方便查找。

所以情况就是这样：我选择了这些名字，不是因为它们以前很流

行，而是因为它们有意义。这些过程模式的名字必须创建一种思考模型，该模型可以明确地指出那些重要的事，并且减少困惑。我希望你能明白这点，并接受这个新的术语。

目录

Part 1 第一部分 开始

本部分内容
第1章 主要优点
第2章 关键过程模式
第3章 活文档
第4章 开始改变

第1章 主要优点

在互联网时代，交付速度是当今软件开发的主题。十年前，项目通常要持续好几年，并且项目阶段是以月来衡量的。如今，多数团队的项目周期是按月来衡量的，而项目阶段则减少到几周甚至几天。任何需要长远规划的东西都将被抛弃，比如大量的前期软件设计和详细的需求分析。超过项目阶段平均周期的任务将不复存在。跟代码冻结(Code Freeze)以及数周的手动回归测试说再见吧！

变化频率如此之高，文档很快就会过时。不断更新详细需求说明和测试计划(Test Plan)需要投入大量精力，相当浪费。那些以往在日常工作中依赖于此的人们，如业务分析师或者测试人员，在这个每周迭代的新环境中经常会无所适从。开发人员原本以为不会受到纸质文档缺失的影响，现在却要把时间浪费在不必要的返工与功能维护上。他们不是花时间去制订宏伟的计划，而是要浪费数周的时间去修正有问题的产品。

在过去的十年里，软件开发社区致力于使用“正确”的方式来构建软件，关注使用技术实践和思想来确保质量。但是，正确地构建产品和构建正确的产品是两码事。我们要二者兼顾才能取得成功。

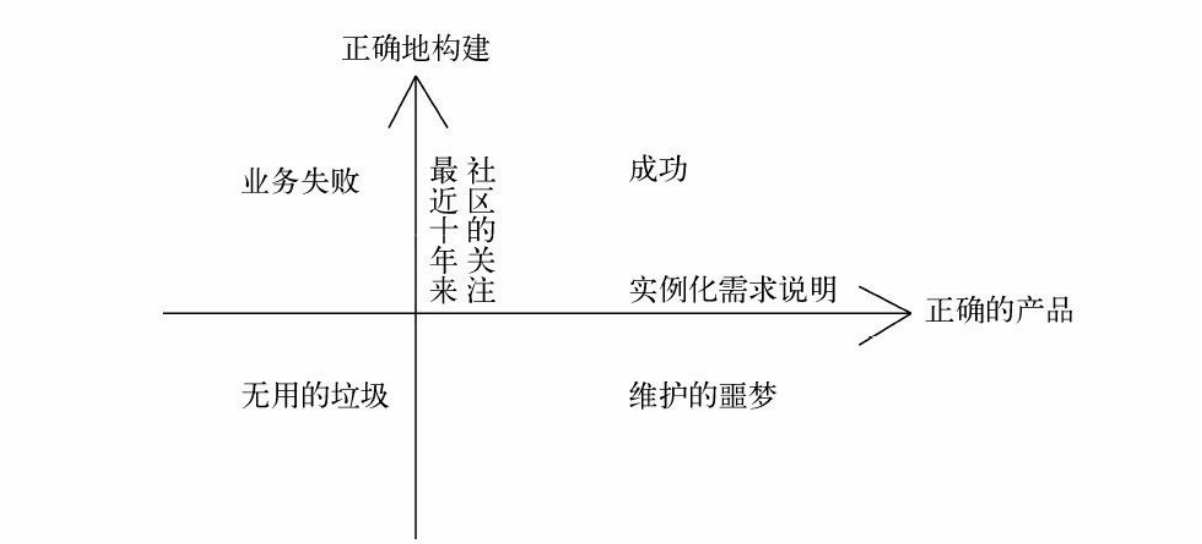


图1-1 实例化需求说明可以帮助团队构建正确的软件产品，而技术实践可以确保正确地构建产品

想要有效地构建正确的产品，软件开发实践必须满足以下几点。

保证所有项目干系人和交付团队的成员都对需要交付哪些东西有一致的理解。

有准确的需求说明，这样交付团队才能避免由模棱两可和功能缺失造成的无谓返工。

有用来衡量某项工作是否已经完成的客观标准。

具有引导软件功能或团队结构变更的文档。

传统意义上，构建正确的产品需要庞大的功能需求说明、文档以及漫长的测试阶段。如今，软件每周都要有交付，这一套已经行不通了。我们寻求的方案要能带来如下好处。

避免过度说明需求从而产生浪费，避免花时间在开发前会发生改变的细节上。

有一种可靠的文档，可以解释系统的行为，据此我们能容易修改系统行为。

可以有效地检查系统行为与需求说明的描述是否一致。

以最少的维护成本维持文档的相关性与可靠性。

适合短迭代和基于流的过程，这样能为即将开展的工作提供即时足够的信息。

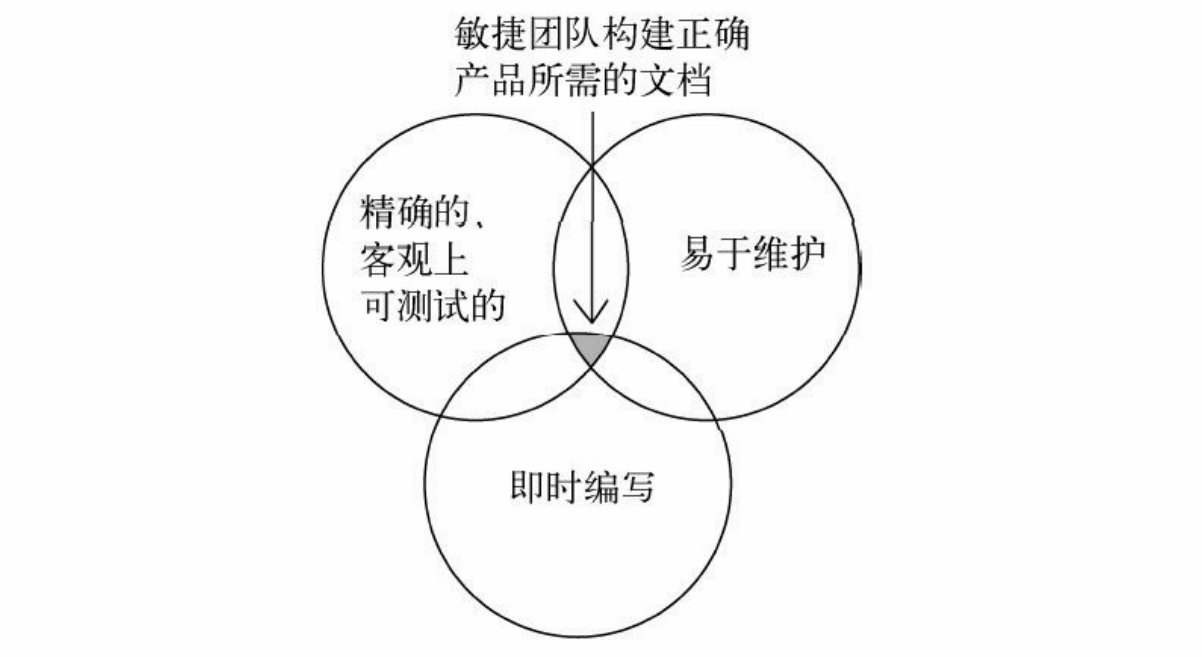


图1-2 对于敏捷项目，构建正确文档的关键因素

乍一看，这些目标似乎互相冲突，但有很多团队已经成功地达成了所有目标。在为本书做调研时，我采访了30个团队，他们完成了大约50个项目。我试图找出一些模式与通用做法，并挖掘出这些方式背后的基本原则。这些项目的共同思想，定义了一种构建正确软件的好方法：实例化需求说明。

实例化需求说明是一组过程模式，它帮助团队构建正确的软件产品。使用实例化需求说明，团队编写的文档数量恰到好处，在短迭代或基于流的开发中可以有效地协助变更。

实例化需求说明的关键过程模式将在下一章介绍。本章我将阐述实例化需求说明的好处。我将使用实例化需求说明的风格来进行阐述，而不是以理论介绍的方式来构建一个案例，我将展示18个真实的例子，它们都来自于那些大大受益于实例化需求说明的团队。

在开始之前，我想强调一下，在一个项目中很难孤立地看待某种思想的影响或作用。本书所描述的实践，可以与已经开展的敏捷软件开发实践[例如测试驱动开发(TDD)、持续集成以及使用用户故事做计划等]共同使用，而且可以增强其他实践的效用。当我们转而去看那些有着不同背景的项目时，很多模式浮现了出来。我采访的团队中，有些在实施实例化需求说明前一直使用敏捷过程，而有些团队则是在过渡到敏捷过程的过程中实施了实例化需求说明。大多数团队使用基于迭代的过程，例如Scrum和极限编程，或者是基于流的过程，例如看板。但是有些团

队，尽管他们使用了这些实践，但他们的过程以任何标准来看都不是敏捷的过程。然而，他们大多都收获了如下类似的收益。

更有效地实施变更。他们拥有活文档——系统功能的可靠信息来源——让他们得以分析潜在变更的影响，同时可以有效地分享知识。

更高的产品质量。他们清晰地定义了预期，使得验证过程很有效率。

更少的返工。他们在需求说明上协作得更好，并确保所有团队成员对预期达成共识。

同一项目不同角色的活动协调得更好。改善协作形成定期的交付流程。

在接下来的4个小节中，我们将通过现实世界的例子，近距离地审视这些收益。

1.1 更有效地实施变更

在为本书做调研的过程中，我获得的最重要的经验是关于活文档(living Documentation)的长期收益的——事实上，我认为这是本书的一个最重要信息，本书广泛地涵盖了这部分内容。活文档是系统功能的一个信息源，它与程序代码一样可靠，但更容易使用和理解。活文档帮助团队共同分析变更所带来的影响并讨论潜在的方案。团队还可以为新的需求扩展已有的文档。长此以往，可以使需求说明和实施变更更有效。大多数成功的团队都发现活文档的长期收益是实施实例化需求说明所带来的结果。

总部设在美国西得梅因市的爱荷华州助学贷款流动资产管理公司(Iowa Student Loan Liquidity Corporation，下文简称Iowa Student Loan)，在2009年进行了一项相当重要的商业模式变更。过去一年，金融市场动荡使得贷款方几乎无法为私人学生贷款找到资金来源，因此，许多贷款方被迫放弃私人学生贷款市场或改变自己的商业模式。该公司适应了当时的市场。它从银行和其他金融机构募集资金来支助私人助学贷款，而不是使用债券收益。

Tim Andersen是一位软件分析师，同时也是一名开发人员，他说为了有效地适应市场，他们不得不“有声有色地进行系统核心大检修”。在开发软件时，他们的团队把活文档作为一项主要机制来编写业务需求文档。活文档系统让他们可以探悉新需求所带来的影响、帮助他们确定所需的变更，而且可以确保系统的其余部分仍旧正常工作。他们当时只花了一个月时间就对系统实施了根本性的变更并将其发布到了生产环境，活文档系统是做这项变更的根本。Andersen说：

“任何未进行这些测试（活文档）的系统，都必将导致开发停顿和重写。”

在加拿大魁北克省的蒙特利尔市，Pyxis技术公司的Talia项目团队也有类似的经验。Talia是企业系统的一个虚拟助理，它是一个拥有复杂规则、能与员工交流的聊天机器人。从最初开始，Talia团队就使用实例化需求说明来构建一个活文档系统。一年之后，他们不得不从头开始编写虚拟代理引擎的核心——而此时，正是在活文档方面的投资大显成效的时候。Talia的产品总监André Brissette是这样说的：

“如果没有活文档，任何重大重构都无疑是自寻死路。”

他们的活文档系统使得团队在变更完成时可以自信地说，新系统具有和老系统一样的功能。该活文档系统还能帮助Brissette管理并追踪项目的进度。

总部位于伦敦的现场音乐消费性网站Songkick的团队在重新开发网站活动摘要时，使用了一个活文档系统来协助变更。他们意识到目前的摘要系统无法扩展到所需的容量，活文档在重新构建摘要系统时就提供了有力的支持。Phil Cownas是Songkick的CTO，据他估计，因为拥有了活文档系统，他们的团队在实施变更时节省了至少50%的时间。据Cowans所述：

“因为我们拥有让人满意的覆盖率，并且我们确实信任这些（在活文档系统里的）测试，所以我们很有信心可以快速地对基础结构进行大的变更。我们知道，系统功能不会改变，即使变了，测试也会发现。”

ePlan Services是一个养老金服务机构，位于科罗拉多州的丹佛市，它的开发团队从2003年开始就已经使用了实例化需求说明。他们构建并维护一个金融服务系统，该系统涉及众多的项目干系人、复杂的业务逻辑以及复杂的监管需求。在项目开始三年之后，其中一位经理搬去了印度，而对于系统遗留部分，有些内容是只有他才掌握的。根据ePlan Services的测试人员及Agile Testing:A Practical Guide for Testers and Teams一书作者Lisa Crispin的描述，当时，团队努力地学习那位经理所拥有的知识并将其构建成活文档。活文档系统帮助他们获得了业务流程的专业知识，并立即提供给所有的团队成员。他们借此消除了知识传递的瓶颈，可以有效地支持并扩展系统。

注释：①该书中文版名为《敏捷软件测试：测试人员与敏捷团队的实践指南》，已由清华大学出版社于2010年出版。——译者注

在比利时Oostkamp的IHC集团，病人管理中心项目组实施了一个活

文档系统，并取得了类似的结果。该项目开始时重写了一个大型机系统，它是从2000年开始的，目前还在进行中。Pascal Mestdach是该项目的方案架构师，他说团队从中受益匪浅：

“当时遗留系统中的一部分功能只有少数几个人了解，而现在情况已经好很多了，那是因为团队拥有一套针对那部分功能的、不停增长的测试套件（活文档），它描述了该遗留系统的功能。当专家休假时，还可以从活文档系统中寻找问题的答案。对其他开发人员来说，可以更清晰地了解软件中某部分的功能。并且还是测试过的。**”**

这些例子阐述了活文档系统如何帮助交付团队分享知识并应付人员变动。它还使得业务可以更有效地响应市场变化。我将在第3章里对此做更具体的说明。

1.2 更高的产品质量

实例化需求说明可以改善交付团队成员之间的协作，促进商业用户更好地参与，并为交付提供清晰客观的目标——大幅提高产品质量。

有两个突出的案例，分别来自Wes Williams[来自世博控股(Sabre Holdings)的敏捷教练]以及Andrew Jackman[为法国巴黎银行(BNP Paribas)的一个项目工作的顾问开发人员]，他们将描述之前失败过多次的项目如何通过实例化需求说明走向成功。本书中描述的方法帮助他们的团队克服了业务领域的复杂性，之前这种复杂性是很难处理的。同时还帮助他们确保了交付的高质量。

在世博控股，Wes Williams工作的项目是一个为期两年的航班订票项目，团队分布在全球各地，流程又是数据驱动的，这使得项目十分复杂。项目有3个团队，30名开发人员，分布于两个洲。据Williams说，系统头两次构建都失败了，但是第三次使用实例化需求说明后就成功了。Williams说：

“我们在一家大客户（一家大型航空公司）上线时缺陷非常少，在（业务验收）测试阶段只有**1**个缺陷是比较严重的，是故障切换**(fail-over)**相关的问题。**”**

Williams认为实例化需求说明是他们取得成功的一个关键因素。除了保证高质量外，实例化需求说明还有助于建立开发人员和测试人员之间的信任。

在法国巴黎银行，Sierra项目是另一个很好的例子，可以展现实例化需求说明如何带来高质量的产品。Sierra是一个债券的数据仓库，整合了一些内部系统、评级机构和其他来自外部的信息，并将它们分发给银行内部的各种系统。许多系统和组织使用相同的术语，表达的意思却

不尽相同，这导致了许多误解。最初两次实现系统的尝试都失败了，据Channing Walton说，团队中的一个开发人员促使了第三次尝试的成功。第三次努力的成功部分归功于实例化需求说明帮助团队处理了复杂性问题，并且确保了团队的共识。最终的产品质量令人印象深刻。项目从2005年上线以来一直在运行，Sierra项目的顾问开发人员Andrew Jackman说：“生产环境中没有出现大的问题。”现在Sierra项目中的大多数工作人员都不是项目启动时的那些人，但是质量水平一直都很高。

Bekk咨询公司在为一家大型法国银行支行开发租车系统时也取得了类似的成果。Aslak Hellesøy曾是那个团队的成员，还是Cucumber——一个支持实例化需求说明的热门自动化工具的创造者，据他说，尽管现在维护这个软件的是一个全新的团队，但他们在系统上线后的两年中却只发现了5个缺陷。

Lance Walton曾在一家大型瑞士银行伦敦分行的一个项目中担任过程顾问，这个项目是要开发一个订单管理系统，开始的几次也都失败了。Walton进入这个项目时，大家都认为实现这个系统需要至少和开发团队一样大的支持团队。他的团队使用了实例化需求说明，项目开始9个月后就交付了生产系统，一天内就通过了业务验收测试，之后6个月内没有发现任何缺陷。Walton说新的系统不需要额外的支持人员，成本比预期要低，而且团队更早地交付了成品。相比之下，他们旁边的团队需要10倍于开发团队的支持人员。Walton指出：

“现在团队依然每周发布一次，用户总是对它非常满意。从质量上看，它棒极了。**”**

实例化需求说明的技术不仅仅适合于新建项目，同时也适用于改建项目。建立起值得信赖的文档、清理遗留的系统，都需要一定的时间，但是团队很快就能看到诸多的好处，并对新的交付充满信心。

还有一个不错的例子是伦敦摩根大通的外汇交易系统。Martin Jackson是该项目的自动化测试顾问，他说业务分析员预计项目会推迟，然而事实上，项目提前两个星期就交付了。高质量的产品让他们成功地在一个星期内完成了业务验收测试阶段，而不是原先计划的4个星期。Jackson说：

“我们部署好系统后，系统工作正常。业务人员向董事会报告说这是他们经历过的最好的用户验收测试**(UAT)**。**”**

实例化需求说明还使Jackson的团队在项目开发晚期快速实现了“一次重大的技术改动”，提高了计算的精确度。Jackson称：

“FitNesse套件（活文档）覆盖的所有功能，通过了完整的系统测试和用户验收测试，在生产环境上线时也没有发现任何缺陷。系统测试时发现了几个核心计算组件以外的错误。业务人员之所以觉得用户

验收测试非常好，是因为出现计算错误时，我们都非常确定根本问题是在计算代码的上游。使用了**FitNesse**后，很容易诊断出缺陷的根源，从而可以更加利落快速地交付到生产环境中。”

科罗拉多州丹佛市的惠好公司有个软件开发团队，他们编写并维护一些工程应用和木制框架的计算引擎。在使用实例化需求说明以前，结构工程师通常不会参与到软件开发过程中，即使团队正在处理一些复杂的科学计算公式和规则。这导致了一些质量问题和延误，由于使用这个引擎的应用程序有好几个，计算过程变得更加复杂。项目的软件质量保证主管Pierre Veragen认为发布前的艰难时期会拖累项目，版本发布出去后很少会没问题。

实施实例化需求说明后，团队现在和结构工程师合作制定需求说明，并自动化验证结果。当有变更需求进来时，测试人员和结构工程师一起得出期望的计算结果，并在开发开始前用实例把结果记录在需求说明中。之后批准变更的工程师会编写需求说明和测试。

Veragen说新方法的主要好处是他们在做改动时有信心了。到2010年初，他们的活文档系统中已经有超过30000个检查，而且几年内都没有发现大的缺陷，现在已经停止追踪缺陷了。Veragen指出：

“我们不需要（缺陷数）这个指标了，因为我们知道它不会再回来了……工程师们喜欢测试先行的方式，并且能直接访问自动化测试。”

Lance Walton参与过一家大型法国银行伦敦分行的信用风险管理程序的开发。项目刚开始的时候，有外来的顾问帮助团队采用极限编程的实践，但是他们没有采用任何实例化需求说明的做法（虽然极限编程包括客户测试，这个与可执行的需求说明很接近）。6个月后，Walton加入了这个项目，他发现代码质量很低。虽然团队每两个星期都会有交付，但是写出来的代码使验证变得很复杂。开发人员只测试最近实现的功能，随着系统的增长，这样的做法就不够了。“当有版本发布时，大家都紧张地围坐着，想确保所有功能都能正常运行，并且期望可以在几个小时内发现一些问题。”Walton如此说。在实施实例化需求说明后，产品的质量和人员的信心都有了显著的提高。他补充道：

“我们十分确信我们发布的版本没有问题。我们高兴地部署完以后就出去享受午餐了，不用再担心是否会出问题。”

与此形成鲜明对比的是，英国贸易者传媒(Trader Media)集团的网站重写项目停止使用实例化需求说明后，却遭遇了质量问题。起初团队协作完成需求说明和自动化验证。在管理层的压力下，他们为了更早更快地交付更多的功能而没有继续下去。测试团队的主管Stuart Taylor说：“我们注意到质量出现了大幅下滑……以前我们（测试人员）很难找到缺陷，而后来我们却发现一个用户故事会有四五个缺陷。”

并不局限于敏捷团队

不是只有敏捷团队才可以从协作制定需求说明中获益。在**Bridging the Communication Gap**一书中，我建议在更为传统的结构过程中应用类似的实践。在那本书出版后，我在为本书做调研时正好碰到一家公司这么做了。

英国**Sopra**集团的高级测试顾问**Matthew Steer**帮助一个大型电信公司的第三方离岸软件交付伙伴实现了这些实践。他们意识到项目需求定义不明确后，决定作出改变。**Steer**比较了实施实例化需求说明前后一年的交付成本。不出意料，这些项目使用瀑布方式开发，没能达到零缺陷的级别，但是这些改变**“**提高了上游缺陷的发现率，减少了下游的返工和成本**”**。**Steer**说：

“我们在软件生命周期早期发现的很多缺陷，传统上要到晚期才能发现，这足以证明这个方法行之有效。缺陷数在生命周期的晚期有明显的下降，而在早期则有所提升。**”**

最后结果是，交付成本仅在**2007**年就节省了**170**万英镑。

1.3 减少返工

一般来讲，频繁地发布会促进快速反馈，使得开发团队能够更快地发现错误、修复错误。但是快速迭代并不能避免错误。通常情况下，团队实现一个功能时会有三四次反复。开发人员称，这是因为客户在拿到产品试用前并不知道自己想要什么。我并不这么认为。使用实例化需求说明后，通常团队第一次实现的就是客户所要的，无需返工。这可以节省大量的时间，并使得交付流程更具可预测性、更加可靠。

位于伦敦的英国天空广播公司(British Sky Broadcasting)的天空网络服务(SNS)部门负责宽带和电话的服务配置(provisioning)软件，它的业务流程和系统集成都极为复杂。该部门由6个团队组成，他们使用实例化需求说明已经有好几年了。据他们的资深敏捷Java程序员Rakesh Patel说：“当我们说交付时，确实是能马上交付的。”并且该部门在Sky公司内具有很高的声望。Patel曾和其他公司的团队一起工作了一段短暂的时间，他对两个团队做了比较，他说：

“他们（其他公司的程序员）每次在迭代**(sprint)**快要结束的时候才把软件交给测试人员，测试人员总是发现问题并退回给程序员。而在这里**(Sky)**，我们不会如此反复。如果有错误，我们会编写一个测试，而后在开发过程中使测试变绿——要么通过，要么不过。我们可以当场发现问题。**”**

其他不少团队注意到了返工的大量减少，其中包含LeanDog，它为

一家美国大型保险机构开发聚合应用软件。他们的应用软件为很多大型主机和基于Web的服务提供统一的用户界面，而且由于拥有来自全国各地的大量项目干系人，该软件变得更加复杂。最初，在需求方面，该项目遭受了很多功能缺失的问题。Rob Park是LeanDog里帮助团队转型的敏捷教练，他说：

“刚开始理清头绪时，我们需要澄清需求，而后我们发现不得不切实做些改变。”

该团队实施了实例化需求说明，结果需求说明改善了，返工也减少了。据Park说，虽然当程序员针对某个故事卡开展工作时，有些问题还要向业务分析师咨询，但是“问题已经大为减少，而且重复性工作少了，只剩下不同的问题”。对他来说，实例化需求说明最有价值的方面在于“当着手实现一个故事时，你可以领会它的意图，并了解它的范围。”

很多团队还发现在开发周期的起始阶段，使用实例化需求说明会让需求更加精确，这样管理产品功能清单(product backlog)会更加容易。例如，能够尽早识别太含糊或有太多功能缺失的故事，这样可以防止以后出现问题。如果没有实例化需求说明，团队经常要到迭代中期才发现问题，这会中断流程而且需要耗费时间重新讨论——在大公司，决定功能范围的项目干系人往往无法轻易预约到。

实例化需求说明能帮助团队建立一个协作制定需求的过程，这可以减少迭代中期的问题。此外，实例化需求说明适用于短迭代，并且不需要花费数月的时间来编写冗长的文档。

Ultimate软件公司位于佛罗里达州的韦斯顿，对于它的全球智能管理(Global Talent Management)团队来说，减少返工是一个主要的优点。协作制定需求说明在专注开发工作方面有着显著的影响。据Ultimate软件公司的资深测试开发工程师Scott Berger所述：

“在团队认可一个故事之前，与产品负责人一起审核测试场景可以使工作小组（产品负责人、开发人员和测试人员）澄清模棱两可或缺失的需求。有时，会议结果甚至会把故事给撤销了，例如，测试场景会揭露出系统隐藏的复杂性或相互矛盾的需求。有一次，进行这样的讨论之后，大家做出的决定是几乎重新设计整个功能！产品负责人获得了重写和重新分割需求的机会，而不是在开发进行之后，中途停止或取消该故事。通过举行这些会议，我们发现自己的生产力和效率都提高了，因为减少了浪费，而且模糊和需求缺失的程度降到了最低。同时还让团队对预期达成了共识。”

大多数团队显著地减少或完全消除了由于误解需求或忽视客户的期望而造成的返工。本书所描述的实践，可以让团队更好地与商业用户打

交道，并确保大家对结果达成共识。

1.4 更好的协作

实例化需求说明的另一个重要好处是能够使得不同的软件开发活动在短迭代周期里更好地协作。根据我的经验和本书中的案例研究，对很多团队来说，采用Scrum最大的绊脚石就是无法彻底完成迭代里的任务。很多团队还坚持“老旧的”理念：先完成开发，然后完成测试，最终修饰润色后发布。这会滋生一种错觉，以为开发是分阶段完成的；而事实上，开发的完成是需要后续的测试和修复的。在Scrum进度板上，“done”列代表开发人员觉得有些事情已经完成，“done-done”列意味着测试人员赞同任务已经完成，等等之类（甚至有人说他们采用了“done-done-done”列）。我们的工作往往会落入这种模式，同时测试的结果影响到下一个周期，这会造成很大的变数并使得交付过程变得更难预测。

实例化需求说明解决了这个问题。本书所介绍的实践让团队清晰地定义一个可以普遍理解和客观衡量的目标。因此，很多团队发现他们的分析、开发和测试活动变得可以更好地协作了。

uSwitch是一个成功改善协作的例子——它是英国最繁忙的网站中的一个。从前，uSwitch很难知道某个功能何时可以完成，因此他们在2008年实施了实例化需求说明。该网站的一位开发人员Stephen Lloyd说：“过去，我们将完成的工作交给QA部门，而他们很快会告诉说我们忘记了测试某个场景，这给我们造成了很多问题。”通过实施实例化需求说明，他们克服了这个问题。Lloyd说他们现在更好地整合成了一个团队，并且对业务需求有了更好的理解。过程的改变还提高了软件的质量。该网站的另一名开发人员Hemal Kuntawala则这么说：

“我们整个网站的错误率显著地下降了，修复问题也比以前快很多。如果在线的网站确实出现了问题，我们通常能够在几个小时内修复，而此前需要几天甚至几周。”

Beazley是一家专业保险公司，他们的团队也经历了协作改善的过程。他们在美国的业务分析师和在英国的开发人员、测试人员一起工作。他们实施实例化需求说明主要是为了确保软件能够在迭代结束时完成。Ian Cooper是Beazley一个开发团队的队长，他说：

“我们一直都写单元测试，但问题是当中有一条鸿沟——这些测试可以告诉我们软件是否工作，却没有告诉我们它是否是客户所需的。过去我们甚至没有让测试人员在（开发的）同一个周期内进行测试。他们（测试人员）会把前一轮迭代的信息反馈到当前的迭代里。不过

现在已经没有这种情况了。我们对验收有了更清晰的认识。”

AdScale.de是一个在线广告市场，他们在新西兰的团队有类似的经历。在项目启动两年之后，日益复杂的用户界面和系统集成使得代码库变得非常大，仅仅使用单元测试已经无法有效地进行管理。开发人员认为事情已经完成了，于是继续去做其他事情，然而在测试人员评审后，开发人员却不得不进行返工。因为测试和开发之间的脱节，找到问题所在需要花费很长的时间。前面几轮迭代的问题影响了后面的迭代，扰乱了开发的流程。在实施实例化需求说明后，开发和测试的协作更紧密了。Clare McLennan是这个项目的开发/测试人员，她说：

“即时反馈减少了发布过程的很多压力。以前，开发人员对我们感到沮丧，因为他们的功能无法发布。同时我们也对他们很无语，因为他们没有修复问题，导致我们无法测试他们的功能。我们在等他们而他们也在等我们。现在这个问题已经不存在了，因为全部测试一次只需一个小时。前一轮迭代完成的功能不会被退回而放到下一轮迭代里。”

实例化需求说明使得团队可以用一种清晰、客观和可衡量的方式定义预期的功能。它还能加速反馈、改善开发流程，并防止中断计划好的工作。

1.5 铭记

正确地构建产品和构建正确的产品是两回事。二者兼顾才能取得成功。

实例化需求说明能够在适当的时间提供恰好够用的文档，帮助使用短迭代或基于流的开发过程构建正确的产品。

实例化需求说明有助于提高软件产品的质量，显著减少返工，并使得团队更好地在分析、开发和测试活动中进行协作。

从长远来看，实例化需求说明有助于团队创建一个活文档系统，它是一种具有相关性的、可靠的功能描述，会随着程序代码的变更自动更新。

实例化需求说明的实践配合短迭代(Scrum、极限编程)或基于流的开发方法（看板）一起使用，效果最好。有些思想也适用于结构化开发过程[统一软件过程(RUP)、瀑布过程]，而且已经有一些案例表明，有公司因此节省了上百万经费。

第2章 关键过程模式

实例化需求说明是一组过程模式，它可以协助软件产品的变更，确保正确的产品能够有效地交付。在为本书做调研的过程中，我采访了很多团队，图2-1展示了那些最成功的团队普遍采用的一些关键模式，以及这些模式之间的关系。这些团队大多是在寻找更高效的方法去构建和维护软件的过程中，通过反复的试验和摸索实现了新的过程思想。我希望，通过揭示他们过程中的模式，能够帮助其他团队从容地实施这些想法。

为什么需要模式

本书提到的过程思想形成了一些模式，也就是说，它们是可以再现的，可以为不同的团队所使用；我说的不是克里斯托弗？亚历山大（著名建筑师）对模式的定义。我所说的过程思想，可以出现在不同的背景中，但它们会产生类似的结果。我并没有像更为传统的模式书籍那样，记录模式是如何创建的，以及它会带来什么样的影响。敏捷联盟功能测试工具**(Agile Alliance Functional Testing Tools)**讨论组组织了多次模式编写的工作坊，想以一种更加传统的方式去记录并建立一个模式目录，他们做这项工作的部分原因正是由于本书的案例所取得的成果，但是完成这项工作尚须时日。我决定暂不把模式扩展成为更传统的格式，这项工作留待本书的后续版本去完成。

在Bridging the Communication Gap一书中，我主要关注的是实例化需求说明的实际产出，例如需求说明和验收测试，忽视了团队在各种不同情况下，可能需要完全不同的方法来获取同样的工件。在本书中，我关注于过程模式、如何创建工件以及它们在流程中如何对后续的工件作出贡献等方面。

即时**(Just-in-time)**

如图**2-1**所示，成功的团队不会一次性或针对所有需求实现整个系列的工作，尤其不会在开发启动前实现。相反，当团队准备好接受更多的工作时，例如在项目开发阶段或里程碑的一开始，他们会从目标中获取范围。只有当团队准备开始实现某个功能时，比如在相关迭代的开始，他们才着手制定需求说明。不要误认为图**2-1**中的系列步骤是大型的瀑布式需求说明。

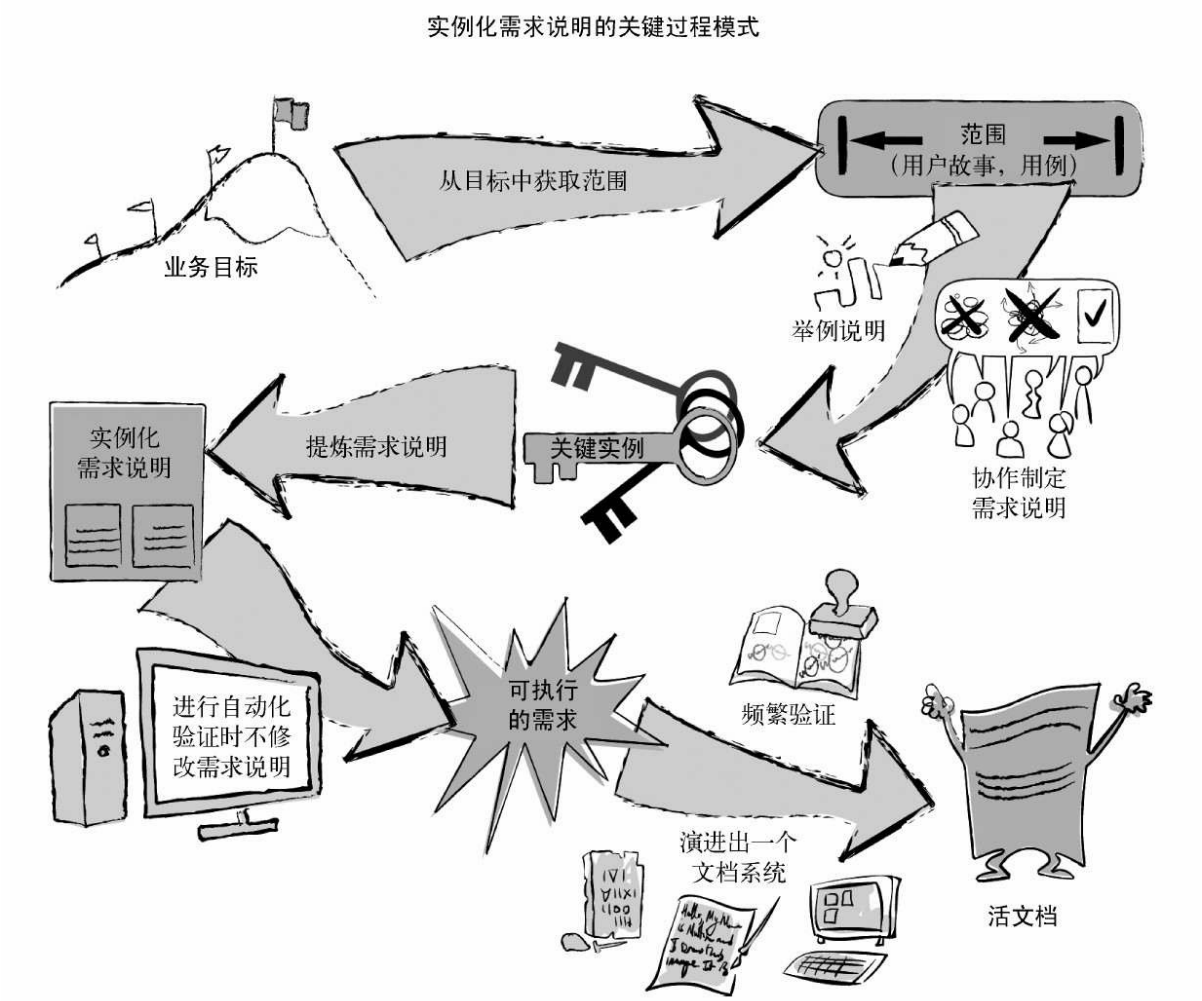


图2-1 实例化需求说明的主要过程模式

本章中，我对关键过程模式提出了一个简要的概述。然后我们将在第二部分仔细讨论实施这些模式的思想和主要挑战。

2.1 从目标中获取范围

实现范围(Implementation scope)含有对业务问题的解决方案或达成业务目标的手段。很多团队在开始实现软件之前（在此之前发生的一切往往被软件开发团队所忽略），期望客户、产品负责人或商业用户来确定工作的范围。在商业用户明确说明他们的需求后，软件交付团队就依此实现。这样本应该会让客户感到满意。但事实上，这正是构建产品开始出现问题的时候。

如果软件交付团队依赖客户给出用户故事、用例清单或其他相关信息，那么他们其实是在让客户设计解决方案。但是商业用户不是软件设

计师。如果我们让客户去界定范围，那么项目就无法从交付团队已有的知识中受益。这样开发出来的软件是客户所要求的，却不是他们真正想要的。

成功的团队不会盲目地接受软件需求，将其作为未知问题的解决方案，相反，他们会从目标中获取范围。他们以客户的业务目标为起始，然后通过协作界定可以实现目标的范围。团队与商业用户一起工作确定解决方案。商业用户专注于传达所需功能希望达到的目的，以及他们期望由此带来的价值，这样有助于所有人了解所需的功能。然后团队提议一个解决方案，这要比商业用户自己想出来的方案更实惠、更快，并且更容易交付或维护。

注释：①关于一些好的例子，请看 http://gojko.net/2009/12/10/challenging-requirements。

2.2 协作制定需求说明

在设计需求阶段，如果开发人员和测试人员都没有参与，那么我们就必须单独将这些需求传达给他们。在实践中，这很容易造成一些误解，在需求传递的过程中会丢失很多细节。结果是，商业用户不得不在软件交付后进行验证，而如果验证失败，那么团队必须回去修改软件。这些都是不必要的返工。

成功的团队不会依赖于某个人独自去收集正确的需求，而是会与商业用户一起协作制定解决方案。不同背景的人们拥有不同的想法，他们会凭借自己的经验来解决问题。技术专家知道如何更好地使用底层的基础架构，或者如何应用新兴的技术。测试人员知道从哪里寻找潜在的问题，而团队则应该去做防止出现这些问题的事情。在设计需求时需要捕获所有这些信息。

协作制定需求说明使我们能够充分利用整个团队的知识和经验。它还创造了需求的集体所有制，让每个人能更多地参与到交付过程中。

2.3 举例说明

自然语言是模棱两可的，而且和上下文相关。仅仅使用此类语言编写的需求，无法为开发或测试提供一个完整明确的上下文。开发人员和测试人员在开发软件及编写测试脚本的时候，必须对需求进行解释，而不同的人可能会对一些棘手的概念有着截然不同的解释。

有些事情看起来简单易懂，但事实上想要完全理解它们，要有领域的专业知识或者专业术语的知识才行，这时尤其容易引发一些问题。理

解上的细微差别会有一个累积效应，往往导致交付后需要返工。这会导致不必要的延误。

在首次使用编程语言实现需求的过程中，成功的团队不会等待需求被精确表述，而会举例说明需求。团队与商业用户一起工作，确定出那些描述预期功能的关键实例。在此过程中，开发人员和测试人员往往会提出一些额外的实例，用于说明边界情况，或重点标识出系统中某些特别有问题的地方。这可以清除功能分歧和不一致的地方，并确保所有参与者都对需要交付的东西有一个共识，避免由误解及解释不到位导致的返工。

如果系统能按照所有关键实例那样工作，那么它就是大家都认同的。需求说明的关键实例有效地定义了软件需要做什么。它们不仅仅是开发的目标，还是检查开发是否完成的客观评价标准。

如果关键实例容易理解和沟通，就可以被有效用作清晰和详细的需求。

2.4 提炼需求说明

协作过程中的开放讨论可以建立大家对相关领域的共识，但是最终得到的实例往往包含很多不必要的细节。例如，商业用户会从用户界面的角度考虑问题，所以他们提供的实例是关于点击链接和填充字段时系统应如何工作的。如此详细的描述会约束系统：详细描述如何实现某个功能，而不去描述需要什么功能，这是一种浪费行为。过多的细节会让实例更难沟通和理解。

关键实例必须精简才有用。通过提炼需求说明，成功的团队能够移除多余的信息并为开发和测试创建一个具体的、精确的上下文。他们以适量的细节来定义目标，以便实现和验证。他们应明确软件该做什么，而不是软件该如何工作。

提炼好的实例可以当作交付的验收条件。只有当所有实例在系统中都可以正常工作时，开发才算完成。为了让关键实例更容易理解，团队要提供一些额外信息，此时，实际上团队就已经创建出了带实例的需求，它是一种工作规范和验收测试，也可用作将来的功能回归测试。

2.5 自动化验证时不修改需求说明

一旦团队对带实例的需求达成一致并且对其进行了提炼，那么团队就可以将它们当作要实现的目标以及验证产品的手段。在开发过程中，这些测试将对系统进行多次验证，以确保它符合目标。人工运行这些检查会产生不必要的延误，并且反馈也会较慢。

快速反馈是短迭代或流程模式软件开发中一个必不可少的元素，所以我们要把验证系统的流程变得廉价而且高效。一个显而易见的解决方案就是自动化。但是在概念上，这种自动化与通常意义上的开发或测试自动化有所不同。

如果我们使用传统的编程（单元）自动化工具或传统的功能测试自动化工具来实现关键实例验证的自动化，那么当业务需求和技术自动化之间丢失细节时，有可能会引入问题。商业用户终究无法获取技术上的自动化需求说明。当需求改变时（是当，不是如果），或者当开发人员或测试人员需要获取进一步澄清时，我们将无法再使用之前自动化的需求说明。我们可以同时用两种形式来保留关键实例：一种是测试，另一种是可读性更强的形式，如Word文档或网页。但只要存在多种版本，我们就会有同步的问题。这就是为什么纸质文档永远不是理想选择的原因。

为了从关键实例中获得最大的收益，成功的团队在做自动化验证时不会去改变需求信息。在自动化过程中，他们几乎完全不改变需求说明——这样就不会有错译的风险。当他们进行自动化验证而不改变需求说明时，关键实例看起来与他们写在白板上的几乎一样：团队的所有人员都可以理解、可以访问。

团队所有人员都可以理解、访问的自动化的实例化需求说明，变成了可执行的需求说明。我们可以把它作为开发的目标，同时可以轻松地检查系统是否按照预期运作，而且我们可以使用同一个文档获取商业用户的澄清。如果我们要变更需求说明，只要在一个地方变更即可。

如果你从未见过用于自动化可执行需求的工具，下面的内容似乎会令人难以置信，但不管怎样，请看图2-2和图2-3。图中展示了两个流行的工具Concordion和FitNesse，它们实现了完全自动化的可执行需求。

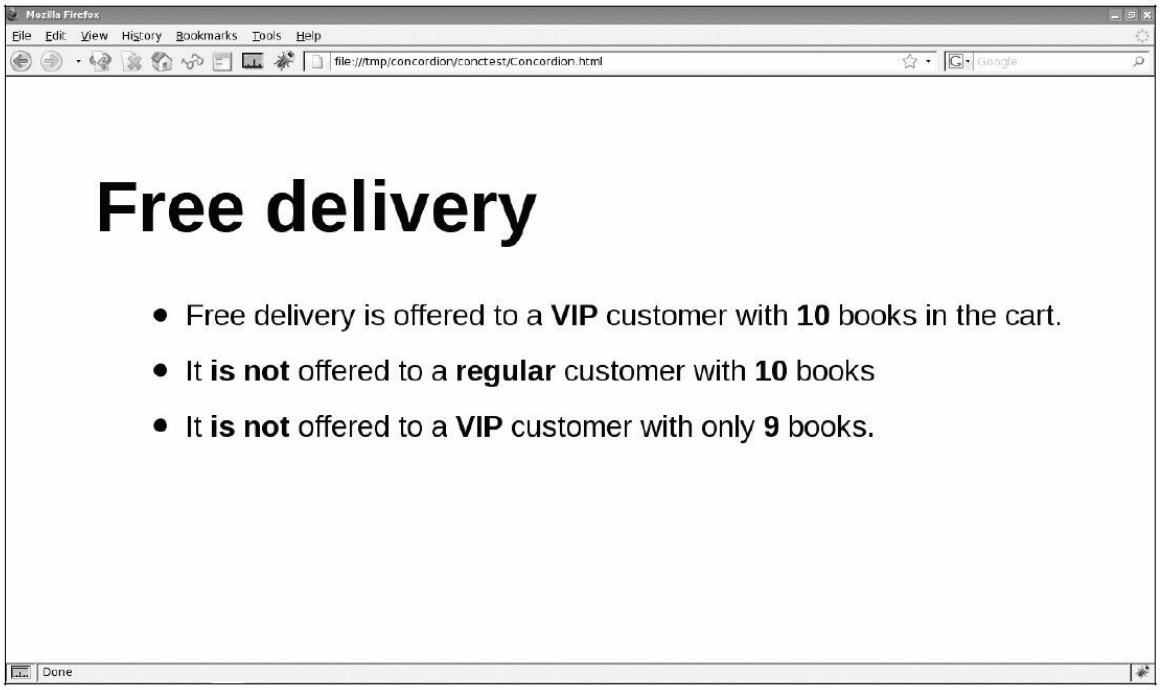


图2-2 一个使用Concordion来自动化的可执行需求

The prize pool is divided among the winners using the following distribution for winning combinations (number of correct hits out of six chosen numbers). Example below is for $2M payout pool.

Prize Distribution for Payout Pool	2,000,000	
Winning Combination	Pool Percentage?	Prize Pool?
6	68	1,360,000
5	10	200,000
4	10	200,000
3	12	240,000

图2-3 一个使用FitNesse来自动化的可执行需求

还有其他许多自动化框架不需要对关键实例进行任何翻译。本书专注于成功团队为实施实例化需求说明所采用的实践，而不是工具。要了

解更多关于工具的内容，请访问http://specificationbyexample.com，你可以在那里下载到介绍最热门工具的文章。同时，请查阅附录里建议资源列表中的“工具”小节。

测试就是需求说明，需求说明就是测试

当使用非常具体的实例描述需求说明时，它也可以用来测试系统。此类需求说明在自动化之后，就变成了一种可执行的验收测试。因为本书只涉及这种需求说明和测试，我将交替使用需求说明和测试这两个词。在本书里，它们之间没有区别。

这并不意味着没有其他类型的测试——例如，探索性测试或可用性测试就不是需求说明。上下文驱动测试的社区试图找出这些测试类型之间的区别，通过使用背景调查来区别哪些是可以被自动化的确定性验证，哪些是需要个人意见和专家洞察力的非确定性验证。至于上下文驱动的语言，本书只涉及设计和自动化检查。通过实例化需求说明，可以让测试人员与团队其他成员一起协作，利用专家的意见和洞察力，设计出好的检查。测试人员不会手动执行这些检查，这意味着他们还有更多的时间做其他类型的测试。

注释：①请看www.developsense.com/blog/2009/08/testing-vs-checking。

2.6 频繁验证

为了有效地支持软件系统，我们需要知道它在做什么以及为什么那么做。在许多情况下，唯一的办法是深入研究程序代码或者找到能帮我们做到这点的人。代码往往是我们唯一能真正信任的东西，而大多数编写好的文档在项目交付前就已经过时了。程序员是知识的权威和信息的瓶颈。

我们可以很容易地针对系统验证可执行的需求说明。如果验证比较频繁，那么我们就可以像信任代码那样信任可执行的需求说明。

通过频繁地检查所有可执行的需求说明，团队能快速地发现系统和需求说明之间的任何差别。因为可执行的需求说明很容易理解，团队可以和商业用户讨论这些改动，并决定如何处理。他们可以不断地同步系统和可执行的需求说明。

2.7 演化出一个文档系统

最成功的团队不会满足于一些频繁验证的可执行的需求说明。他们能确保很好地组织需求说明，让大家很容易找到和获取，并让它们保持

一致。随着项目的发展，团队对领域的理解也在变化。市场机遇也影响着业务领域模型。能从实例化需求说明中获取最大收益的团队会更新他们的需求说明来反映这些变化，演化出活文档系统。

活文档是关于系统功能可靠的、权威的信息源，任何人都可以获得。它和代码一样可靠，但是更容易阅读和理解。支持人员可以用它来查明系统在做什么以及这样做的原因。开发人员可以用它作为开发的目标。测试人员则可以用它来测试。分析功能变更请求的影响时，业务分析师可以从它开始着手。它还提供了免费的回归测试。

2.8 实际的例子

在本书的其余部分，我会专注于过程模式而不是过程的工件。为了更深入地看待问题，确保你理解这些术语，我在书中囊括了一个例子，用来说明整个过程产生的工件，从商业目标到活文档系统。例子的讨论中指明了我会在哪些章节讨论相关内容。

2.8.1 商业目标

商业目标是项目或项目里程碑的潜在原因。它是帮助商业项目干系人（无论是内部的还是外部的）决定投资软件开发的指导性愿景。商业组织应当能清晰地看到这些目标如何赚取、节省或者保护财富。把“提高对现有客户的重复销售”作为商业目标，是一个不错的开始。原则上，目标应该是可以度量的，这样它就可以指导实现。对于“12个月内对现有客户提高10%的重复销售”和“3个月内对现有客户提高500%的重复销售”这两个目标，相应的软件范围很有可能是非常不同的。可度量的目标让我们可以确定项目成败、跟踪进度和更好地排列优先级。

良好商业目标的例子

12个月内对现有客户提高50%的重复销售。

2.8.2 范围

通过应用我在第5章会讲到的实践，我们可以从商业目标中获取实现范围。实现团队和商业投资者一起提出一些想法，然后把它们分成可交付的软件块。

比方说我们发现一个主题故事是关于客户忠诚度计划的，这个故事可以分解成会员忠诚度管理系统的基本功能和更高级的奖励计划。我们决定首先专注在建立一个基本的会员忠诚度管理系统上：客户注册一个VIP计划，VIP客户有资格获得特定物品的免费送货。我们将推迟关于高级奖励计划的讨论。下面是这个例子的范围。

会员忠诚度管理系统基本功能的用户故事

为了能对现有客户做产品直销，作为营销经理，我想让客户通过加入VIP计划注册个人信息。

为了吸引现有客户注册VIP计划，作为营销经理，我要系统为VIP客户提供特定物品的免费送货。

为了节省开支，作为现有客户，我希望能收到特价优惠的信息。

2.8.3 关键实例

通过应用第6章和第7章所讲述的实践，一旦团队开始实现某个特定的功能，我们就可以为特定的范围产生具体的需求说明。比如，当我们开始做范围中的第二项——免费送货时——必须定义好什么是免费送货。在协作讨论过程中，为了避免运送电子产品和大件物品相关的后勤问题，我们决定系统只提供书籍的免费送货服务。因为商业目标是提升重复销售，我们尝试让客户进行2多次购买，“免费送货”变成了“免费为5本或以上书籍送货”。我们要确定好关键实例，比如VIP客户购买5本图书、VIP客户购买5本以下的图书，或者非VIP客户购买书籍。

接着讨论当客户同时购买了书籍和电子产品时该怎么办。有些人建议扩展范围，例如，将订单拆分成两个，只为书籍提供免费送货。我们决定推迟这个决定，先实现最简单的。如果订单中有非书籍的物品，我们就不提供免费送货。我们加入下面这个新的关键实例，之后会再讨论。

关键实例：免费送货

VIP客户购物车中有5本书籍可以获得免费送货。

VIP客户购物车中有4本书籍就不提供免费送货。

普通客户购物车中有5本书籍没有免费送货。

VIP客户购物车中有5台洗衣机时不提供免费送货。

VIP客户购物车中有5本书籍和1台洗衣机时不提供免费送货。

2.8.4 带实例的需求说明

应用第8章中的实践，我们从关键实例中提炼出需求说明、创建出一目了然的文档并将其格9式化成便于今后做自动化验证的格式（如下所示）。

免费送货

当VIP客户购买一定数量的书籍时，提供免费送货。免费送货不提供给普通客户或购买非书籍的VIP客户。

假定至少买5本书才能获得免费送货服务，那么我们会得到以下预期：

实例

客户类型	购物车中的物品	送货
VIP	5本书	免费，标准
VIP	4本书	标准
普通	10本书	标准
VIP	5台洗衣机	标准
VIP	5本书，1台洗衣机	标准

这个需求说明——一目了然的文档——可以用作实现的目标和自动化测试的驱动，这样我们就可以客观地衡量什么时候算完成了。把它作为活文档的一部分，保存在需求说明仓库中。FitNesse的wiki系统或者Cucumber功能文件的目录结构就是这样的例子。

2.8.5 可执行的需求说明

当开发人员开始实现需求说明所描述的功能时，基于需求说明的测试开始时会失败，因为测试还没有自动化，功能也还没有实现。

开发人员会实现相关功能并把它与自动化框架关联在一起。他们使用自动化框架从需求说明中获得输入并验证预期的输出，而不需要实际修改需求说明文档。第9章中的概念和实践将帮助我们有效地自动化需求说明。当验证实现自动化以后，需求说明就变成可执行的了。

2.8.6 活文档

所有已实现功能的需求说明需要频繁地进行验证，一般通过自动化构建过程来实现。这样可以确保需求说明保持更新，同时有助于避免功能退化的问题。团队使用第10章中的实践可以使频繁验证顺利进行。

当实现了整个用户故事的时候，需要有人去做首次验证以确保其已经完成，然后重组需求说明确保它和已实现功能的需求说明是一致的。他们将采用第11章的实践，从需求说明逐步演化出文档系统。举例来说，他们可能将免费送货的需求说明移到送货相关的功能体系中，也可能将它们和其他因素促发的免费送货实例合并在一起。为了更容易地访问文档，他们可能会在免费送货的需求说明和其他送货类型的需求说明之间建立链接。

然后这个循环再次开始。一旦我们需要再次回顾免费送货的规则——比如，在做高级奖励计划，或是扩展功能把带书籍的订单和其他货物订单分离开的时候——我们就可以使用活文档来理解现有的功能并注明需要修改的地方。我们可以使用已有的实例来协作制定需求说明，同时举例说明会更加有效。然后我们会举出另一组关键实例，进一步演进免费送货的需求说明，这部分最终会和需求说明的其他部分合并到一

起。这个循环会不断重复。

现在我已经对关键流程模式做了一个简要的说明，大家可以通过第3章对活文档有一个更深入的了解。在第4章里，我会介绍如何开始采用实例化需求说明，这伴随着实现个人流程模式的思想，将会在第二部分进行介绍。

2.9 铭记

实例化需求说明的主要过程模式是从目标中获取范围、协作制定需求说明、举例说明、提炼需求说明、自动化验证时不修改需求说明、频繁验证以及演进出活文档系统。

对于实例化需求说明而言，功能需求、需求说明和验收测试都是一回事。

其结果是一个活文档系统，它解释系统可以做什么，并且与编程语言代码一样确切和可靠，但更容易理解。

不同背景的团队使用不同的实践来实施过程模式。

第3章 活文档

目前，我们一般认为实例化需求说明的过程和工件有两种流行的模型：以验收测试为中心的模型和以系统行为规范为主导的模型。

以验收测试为中心的模型（通常称为验收测试驱动开发，ATDD或者A-TDD）侧重于自动化测试，并把它作为实例化需求说明过程的一部分。这个模型的主要优点是开发目标更加明确，并且可以防止功能退化。

以系统行为规范为主导的模型（通常称为行为驱动开发或者BDD）侧重于制定系统行为的场景。它的主要工作是通过协作和需求澄清，在项目干系人和交付团队之间建立起共识。该模型认为，通过自动化测试预防功能退化也很重要。

我认为这两个模型不存在哪个更优的问题，不同的模型有不同的用途。首次采用实例化需求说明时，如果一个团队有很多功能上的质量问题，那么以验收测试为中心的模型更加有用。当一切运行顺畅的时候，对于说明软件交付的中短期活动，以系统行为规范为中心的模型会比较有用。

在这两个模型中，通过自动化测试预防功能退化是实例化需求说明的主要好处，而且这种好处具有长期性。尽管回归测试的确很重要，但我并不认为这种长期好处是它产生的。首先，实例化需求说明并不是唯

一可以防止功能退化的方法。例如，uSwitch的团队在首次实现了相关功能后，禁用了许多测试（详见第12章），但他们仍然保持着很高的质量。其次，Capers Jones在Estimating Software Costs一书中指出，通过回归测试移除缺陷的平均效率仅有23%。这无法证明成功的团队在实现实例化需求说明上作出长期投资是正确的。

注释：①参见Estimating Software Costs:Bringing Realism to Estimating一书(McGraw-Hill，2007年)第509页。中文版名为《软件项目估计》，由电子工业出版社于2008年出版。同时请参考http://gojko.net/2011/03/03/simulating-your-way-out-of-regressiontesting。

在为编写本书做调研时，我曾有幸采访了一些使用实例化需求说明达5年甚至更久的团队。他们的经验，尤其是最近几年的经验，帮助了我从另一个不同的视角——以文档为中心去看待事情。很多我采访的团队意识到，将实例化需求说明的工件作为长期的文档是很有价值的。大部分团队曾尝试各种方法来定义需求说明和测试，在历经数年之后才发现这一点。作为本书的作者，我的一个主要目标就是要展现实例化需求说明最好的工件——活文档。这将有助于读者从容迅速地实现一个活文档系统，而无需耗费几年的时间反复试验。

在本章中，我会介绍文档模型及其好处。这个模型关注业务流程的文档，确保对业务流程长期有效的维护和支持。该模型对确保实例化需求说明发挥长期作用特别有用，它还可以防止许多常见的测试维护实现问题（本章稍后会有更多介绍）。

3.1 为什么我们需要权威的文档

我遇到过太多次这样的情况了，别人塞给了我一本关于系统的厚厚的书，同时告诫我那本书“并不完全正确”。冗长的纸质文档就像廉价的葡萄酒一样会很快过期，如果你在创建一年后再去使用它，会很头痛的。反过来，维护一个没有任何文档的系统同样令人头疼。

我们要了解一个系统是做什么的，这样才能对建议的变更所带来的影响做出分析、提供支持并排除故障。通常，找出系统在做什么的唯一途径是浏览程序的源代码，并反向推导出系统的业务功能。Christian Hassa是TechTalk公司的合伙人，当我为了编写这本书采访他的时候，他把从代码中挖掘系统功能的过程叫做“系统考古学”。他解释了大部分读者都非常熟悉的一种情况：

注释：①TechTalk公司开发了.NET里的BDD工具SpecFlow。——译

者注

“我们有一个项目需要替换一个遗留系统。没有人知道某个计算结果或报表是如何产生的。用户只是一味地使用计算结果并盲目地信任老的系统。从旧的应用程序反向推导出需求真令人恐怖啊，当然了，结果发现老系统的某些逻辑是错误的。”

即便那些没文档的代码是正确的，对于商业用户、测试人员、支持工程师来说，反向工程是一件几乎无法完成的任务。在大部分项目中，甚至一般的开发人员也做不到。显然，反向工程的做法是行不通的，我们需要更好的办法。

良好的文档不仅仅对软件开发非常有用。许多公司都因为对他们自己的业务流程拥有良好的文档而受益颇多，尤其是当越来越多的业务用上技术时。同其他类型的系统文档一样，业务流程的文档很难编写，而且维护成本也很高。

理想的解决方案是一种易于维护并且维护成本较低的文档系统，这样即便频繁改动底层的编程语言代码，它也还能与系统功能保持一致。事实上，任何一种全面的文档都面临着维护成本高昂的问题。以我的经验来看，改动过时的内容不是主要的成本所在，花时间去找出需要改动的地方通常才是成本较高的地方。

3.2 测试可以是好文档

自动化测试的问题则相反。找出所有需要更新的地方很容易，可以频繁地执行自动化测试，任何失败的测试显然不再与底层代码同步。但是，除非测试设计得易于修改，否则在系统更改后去更新这些测试可能会耗费很多时间。以验收测试为中心的方法，其缺陷之一就是会忽略这种影响。

注重编写测试和执行测试的团队往往会编写出不易维护的测试。随着时间的推移，很多问题会迫使这些团队去寻求制定测试并将测试自动化的方法，以便让测试更容易更新。一旦测试变得易于维护，团队就会看到实例化需求说明带来的其他长期收益。Adam Knight的团队效力于RainStor公司——一家位于英国的在线数据存储方案供应商，他们意识到，如果测试能展现出背后的根本目的，那么它们就是易于维护的。他说：

“在开发一套自动化测试时，如果你能正确地将它们建立起来，用它们展现背后的根本目的，那么你就可以将它们作为文档了。我们制作出HTML报表，列出运行过的测试及其目的。调查任何的回归失败都变

得容易多了。你可以更容易地解决冲突，因为不用回头去查看其他文档就能了解其背后的目的。”

我觉得最后一句是最重要的：如果测试很清晰，他们就不必使用任何其他类型的文档了。ePlan Services的Lisa Crispin说，对她而言，最豁然开朗的时刻之一，就是当她理解了把测试作为文档是多么有价值的时候：

“我们获得了贷款的还款，但系统收取的利息金额不正确。我们认为系统有缺陷。我可以查看**FitNesse**测试并输入数值。也许是需求错了，但目前的代码就是这么计算的。这节约了很多时间。**”**

Andrew Jackman说Sierra团队把测试结果作为技术支持的知识库：

“业务分析师总是可以看到这种好处。当有人询问**Sierra**中的某些数据是怎么来的，他们常常只把测试结果的链接发过去即可——那是舍理的文档。我们的需求说明都不放在**Word**文档中。**”**

前文我提到过爱荷华州助学贷款公司的团队，他们使用测试估算业务模型变更的影响，并指导其实现。SongKick公司的团队使用他们的测试来指导系统变更的实现，并节省了大约50%的时间。我还从其他许多团队听到过类似的故事。

当团队使用某种信息去指导开发工作，实施系统支持，或者估算业务变更的影响时，将这种信息称为“测试”会让人产生误解。支持并演进我们系统的并不是测试，而是文档。

3.3 根据可执行的需求说明创建文档

当一个软件系统持续地受到一系列可执行需求说明的验证时，团队可以确信系统会按照需求说明所述的情形工作——或者，换言之，需求说明会持续描述系统的行为。那些需求说明与系统同存，而且它们始终保持一致。由于我们能立刻发现需求说明与系统功能之间的差异，就可以以较低的成本让它们保持一致。爱荷华州助学贷款公司的Tim Andersen曾经说过，他只相信这种文档：

“如果我拥有的文档不是自动化的，我就不会信任它。那种文档是未经实践检验的。**”**

可执行的需求说明创建出文档的主体，一个关于系统功能的权威信息来源不会遭遇“不完全正确”的问题，而且维护成本相对较低。如果实例化需求说明是书中的一些页面，那么活文档系统就是一本书。

团队交付正确产品所需的所有工件都可以用活文档代替，它甚至有助于外部用户手册的编写（尽管不太可能代替它们）。它很适合短周期迭代或者流程化的过程。由于我们可以在构建软件系统的过程中逐步地

制定出需求说明，因此最终产生的文档是增量式的，而且编写成本很低。同时，我们可以构建起业务流程的文档作为支撑系统，用该文档来演进软件并帮助我们经营业务。当有人在编制500页的材料时，没有必要让世界停止6个月。来自Pyxis的André Brissette说这是对敏捷开发最基本的认识之一：

“敏捷初学者会认为敏捷是没有文档的，这不是事实。敏捷建议我们要选择那些有用的文档。对那些害怕没有文档的人而言，这样的测试是一个保护他们自己的绝佳机会，同时可以让他们看到在敏捷过程中仍然是有文档的，而且那并不是两英尺高的一大堆纸，而是一种更轻量级但紧密绑定在实际代码上的文档。当你询问‘你们的系统是否有这种功能’的时候，你没有一份用来记录系统功能的**Word**文档，相反你有一种可以执行的东西，可以证明系统就是按照你的想法在运行。那才是真正的文档。”

大部分实例化需求说明使用的自动化工具已经支持通过网站来管理需求说明，或者把测试结果导出为HTML或PDF的形式，这是构建起一个文档系统的良好起点。我期望未来几年内会出现很多创造性的工具，帮助我们根据实例化需求说明建立起文档。有一个有趣的项目叫Relish，它可以从一些自动化工具中导入实例化的需求说明，通过格式化可以创建出一个文档系统，用起来很方便。请看图3-1。

注释：①请参考www.relishapp.com。

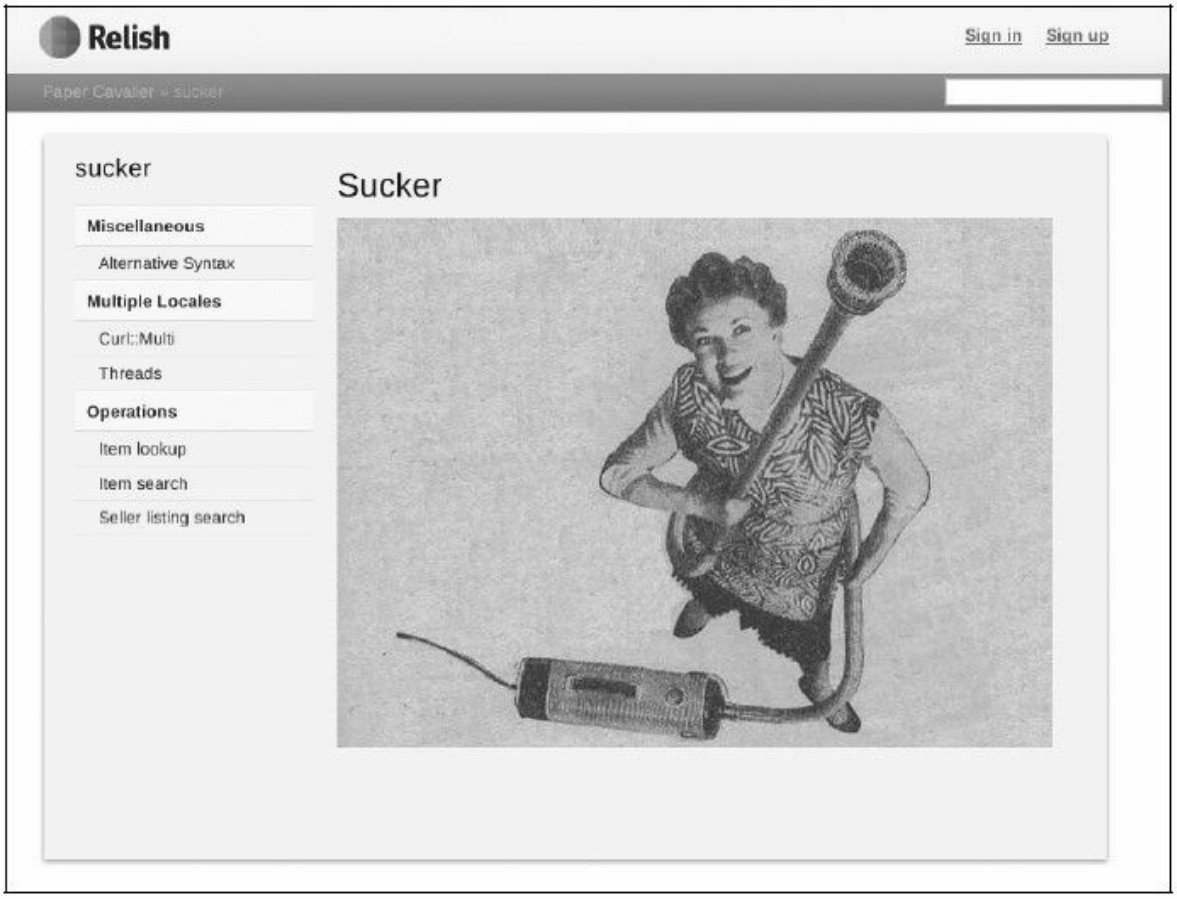


图3-1 Relish根据可执行的需求说明建立起来的文档站点

3.4 以文档为中心的模型所具有的好处

实例化需求说明以文档为中心的模型可以帮助团队避免长期维护可执行需求说明时最常见的问题，同时它也可以帮助团队建立起有用的文档，随着时间的推移这种文档可以促进软件的革新，并避免由于缺乏共享知识而造成的维护问题。

很多我采访过的团队，在持续使用实例化需求说明并用它指导所有工作时，替换掉了他们的系统核心，或者重写了系统的大部分地方。这就是活文档真实的投资回报。无需在系统考古和验证上花费数月的时间，活文档系统已经可以为技术更新或变更提供需求了。

我认为团队应该把活文档看作是单独的工件，与他们交付的系统同等重要。将文档当成关键性交付物是以文档为中心的模型最核心的部分。我觉得这个模型可以解决大部分导致团队使用实例化需求说明失败的常见问题。虽然本书中还没有任何案例可以证明这一点，但我认为这是未来的重要前提。我希望本书的读者以这个不同的视角去看待他们的

过程时，可以更容易、更快速地取得卓越的成果。

比如，明白活文档是一个重要工件后，你立马可以决定是否将验收测试放到版本控制系统里。侧重于业务流程文档可以避免过度关心技术需求说明，还可以保持需求说明从业务的角度去关注系统应该做的事情，而不是去关注测试脚本。清理测试代码不再需要单独的说明。增强测试的结构或者澄清测试意图不再会成为技术负债：它们是标准交付任务的一部分。把验收测试的工作委托给初级开发人员和测试人员是有问题的，这一点突然变得非常明显。有用的文档必定是组织良好的，这会防止团队把成千上万难以理解的测试放在同一目录中。

把活文档看成交付过程的单独工件，团队还可以避免对它投资过度。他们可以事先讨论准备花多少时间去构建活文档系统，以免掉入这样一个陷阱：对测试进行镀金，却牺牲了主要的产品。

我认为需求说明过于抽象可能是文档模型的一个潜在陷阱。我期望这个模型在将复杂业务流程进行自动化的软件系统中能发挥更大的作用。以用户界面为中心的项目可能就不会从中受益那么多，因为这种项目的复杂度不在其底层的业务流程中。

3.5 铭记

实例化需求说明有几种模型，不同的模型有不同的用途。

实例化需求说明允许你渐进性地建立起一个良好的文档系统。

活文档是交付过程的重要工件，与代码一样重要。

侧重于建立业务流程文档系统可以帮助你避免长期维护需求说明和测试造成的大部分常见问题。

第4章 开始改变

实例化需求说明的很多中心思想已经存在几十年了。20世纪80年代晚期，Gerald Weinberg和Donald Gause在《探索需求》一书中谈到了软件需求的沟通问题。这两位作者认为，检查需求的完整性和一致性的最好方式，就是针对它们设计黑盒测试——实际上这就表明了实例化需求说明既是需求说明又是测试的两重性。1986年，德国军方使用了一种手法，用来描述那些为了进行验证，在实施之前就建立验收测试的方法，这种手法后来演变成了V模型。今天，我们使用的是同样的方法，但是把验收测试当作带实例的需求。1989年，Ward Cunningham在WyCASH+项目中应用了举例说明和不修改需求说明进行自动化验证的一些实践。

注释：①Gerald M.Weinberg和Donald C.Gause合著的Exploring Reguirements: Quality Before Design(Dorset House Publishing Company，1989年出版)

注释：②http://fit.c2.com/wiki.cgi？FrameworkHistory

不幸的是，这些思想当时没流行开来。漫长的开发阶段使得它们难以实行。人们得花上几个月的时间，为那些持续数年的项目编写抽象的需求。事先以实例详细说明一切，会使项目延误得更久。

敏捷开发改变了业界对软件交付阶段的看法，并显著地缩短了这些阶段。这让实例化需求说明变得切实可行。迭代和基于流程的项目可以极大地受益于实例化需求说明。以如此短的时间完成一个交付阶段，我们需要尽量消除不必要的工作。需要解决的常见问题是返工、沟通不畅导致的重复工作、为了理解系统而回头阅读代码所浪费的时间，以及手工重复执行相同测试所消耗的时间。使用短迭代或恒定速率的流程进行有效的交付时，需要尽可能多地排除可以预期的障碍，这样，意外的问题才能得以解决。Adam Geras意味深长地说：“高标准就是要为应对寻常问题做好准备，这样你才有时间去处理不寻常的问题。”活文档可以轻易地消除常见问题。

实例化需求说明是解决方案：一种处理寻常问题的手段，这样在几天或几周的软件交付周期内，我们才有更多的时间去处理不寻常的问题。如今，活文档是取得成功的一个必要条件。

本章中，我们将探讨如何着手改变过程和团队文化，以便你去实施实例化需求说明。我们将回顾三个团队的案例研究，它们使用了不同的方法，将协作制定需求说明集成到了迭代和流程式的开发中。最后，我会提出一些有用的想法，让这个过程适用于那些对需求有签收要求和可追溯性要求的开发环境。

4.1 如何开始改变过程

开始改变一个过程从来都不容易，特别是想从根本上改变团队成员之间的合作方式。为了挺过初期的抵制，为将来的变革建立一个成功的范例，大多数团队一开始都会实施一个短期内能改善产品质量或者节省时间的实践。最常见的出发点有以下这些。

如果已经在进行一个过程变更，那么就通过它实现实例化需求说明的主要思想。

将实例化需求说明的思想当作改善产品质量的灵感。

为那些没有自动化功能测试的团队，实施功能测试的自动化。

为那些自动化测试与开发环节相脱离的团队，引入自动化的可执行需求说明。

对那些实践测试驱动开发(TDD)的团队，使用TDD作为下一步的踏脚石。

所有这些出发点都会在短期内产生效益，并能够带来进一步的改善。

4.1.1 把实施实例化需求说明当作更广阔的过程变更的一部分

适用于：新项目

在我采访的团队中，有4个团队在转向敏捷软件开发的过程中，实施了实例化需求说明的核心思想。不用面临别人对于过程变更的抵触，也不用获得管理层的支持。

→实施Scrum、XP或任何其他敏捷过程终归是一种休克疗法，因此如果有可能的话，可以把实施实例化需求说明也纳入其中。

能够做到这一点的团队出现的问题比较少，并且相比那些从功能不健全的Scrum环境开始变更的团队，其过程的实施会更加迅速。主要原因是在敏捷转型过程中，这4个团队都得到了重大的支援（其中3个团队有咨询师在现场，另一个团队的一位成员曾经参与过实例化需求说明的实践）。

4.1.2 专注于提高质量

uSwitch（详见第12章）的团队决定专注于提高产品质量，而不是专注于某个特定过程。他们要求所有队员提出改进建议，并从中找到灵感。最终他们实施了实例化需求说明的大多数过程模式，中间只遇到微

小的阻力。

从管理角度来看，如果团队中有很多人可能会去抵制某个过程变更，那么专注于提高质量就是一个特别好的方法。人们可能会对Scrum、敏捷、实例化需求说明、看板或其他流程相关的事情产生抱怨。公开主动地提高质量是不太可能引起抱怨的。David Anderson提倡在看板里将注重质量作为获得成功的第一步。

注释：①请参阅David Anderson的Kanban: Successful Evolutionary Change for Your Technology Business一书(Blue Hole Press，2010年出版)。

→先找出提高软件质量的最大阻碍，然后解决这个问题。

如果开发人员和测试人员没有一起紧密合作，并且对是否接受某件东西的质量拥有不同的看法，那么把有关产品发布的一些行为展示出来，可能会很有用。在提供电子金融交易服务的LMAX公司，Jodie Parker创建了一个发布候选板，上面显示了3个团队的进度概况，这样所有发布活动都变得可视了。它的作用是显示所有计划交付项的状态、发布的重点、发布前必须完成的一系列任务，以及发布前必须解决的关键问题。所有团队都能看到此类信息，而后提出改善交付流程的建议。

4.1.3 从功能测试自动化开始

适用于：应用到现有项目

我采访的大多数团队都是从功能测试自动化开始采用实例化需求说明的，而后逐渐从先开发后测试转到使用可执行的需求说明来指导开发。对那些已经拥有大量代码并且需要测试人员手动执行验证的项目，这种方式的阻力似乎最小。

有几个团队寻求在测试阶段解决瓶颈问题，结果测试人员必须持续地追赶开发的进度。在短交付周期（几周甚至几天)里，大规模手动测试是不可能的。测试堆积到迭代结束的时候，接着蔓延到下一轮迭代，扰乱了正常的开发流。功能测试自动化可以消除这种瓶颈，并使得开发人员和测试人员共同参与，激励他们参与到流程变更中。Markus Gärtner说：

“对一个遭受‘测试瓶颈’之苦并且不断对抗开发变更的测试人员来说，通过自动化测试来提供有价值的反馈（甚至是在修复缺陷之前）是非常、非常、非常具有吸引力的。这是一个需要努力实现的激励性愿景。”>

→如果你还没有实行功能测试自动化，请记住这是一个容易实现的目标，一个开始实施实例化需求说明的简单方式。

把功能测试自动化作为采用实例化需求说明的第一阶段是很有效的，原因如下。

它带来立竿见影的好处。通过自动化的测试，测试阶段的时间显著减少，遗漏到生产环境的问题也就显著减少。

有效的测试自动化需要开发人员和测试人员的协作，它开始打破这两个群体之间的隔阂。

遗留产品很少有支持简便测试的设计。从功能测试自动化开始，可以迫使团队解决这个问题，让架构更具可测性，同时可以解决有关测试可靠性和测试环境的问题。这能为接下来可执行需求说明的自动化打好基础。

如果大多数测试都是手工的，并且团队以短周期进行工作，那么测试人员往往就是过程中的瓶颈。这使得他们几乎不可能从事任何其他事情。测试自动化让他们有时间去参加需求说明工作坊，并开始尝试其他活动，例如探索性测试。

与手动测试相比，测试自动化可以让团队运行更多的测试，而且运行得更加频繁。这往往可以排除缺陷和不一致的地方，同时透明度的突然提升有助于商业项目干系人看到测试自动化的价值。

编写以及开始实施功能测试自动化往往需要商业用户的参与，他们必须判定某个不一致的问题是缺陷还是系统本身的运行方式。这会带来测试人员、开发人员和商业用户之间的协作。同时，这也要求团队自己找到测试自动化的方法，以便商业用户能够理解，为可执行的需求说明准备好适合的方法。

更快速的反馈有助于开发人员看到测试自动化的价值。

功能测试自动化有助于团队成员理解可执行需求说明自动化所需的工具。

这是否只是在转移工作？

让程序员协作进行测试自动化可以解放测试人员，对此，一个常见的反对意见是程序员因此会有更多事情要做，而这会减慢功能的交付。事实上，业界的总体趋势是团队的程序员比测试人员多，所以将工作从测试人员转移到开发人员并不一定很糟糕——它可能会消除过程中的瓶颈。

实现功能测试自动化将使团队更紧密地合作，并为系统以后使用可执行的需求说明做好准备。为了使这种方式的收益最大化，在实现功能测试自动化的时候，应该使用一个针对可执行需求说明而设计的工具，并使用第9章和第11章的思想做好测试的设计工作。使用传统的“录制播放”测试工具不会带来你所需要的好处。

从系统的高风险部分开始自动化

想要使用自动化测试完全覆盖遗留系统是徒劳的。如果使用功能测试自动化作为实例化需求说明的一个步骤，那么你应该编写足够多的测试来展现测试自动化的价值，并习惯于使用相关的工具。此后，当有需求变更时，就可以开始实现可执行的需求说明，并逐渐增加测试覆盖率。

为了从最早实施的功能测试自动化中获得最大收益，请专注于将系统中存在风险的那部分先自动化掉，这些地方的问题会花费很多财力。防止那些地方出现问题可以立刻体现出自动化的价值。良好的功能测试覆盖率将使团队获得更多的信心。而对风险较小的部分进行自动化所带来的好处可能不值一提。

注释：①详见http://gojko.net/2011/02/08/test-automation-strategy-for-legacy-systems。

4.1.4 引入一个可执行需求说明的工具

适用于：测试人员负责测试自动化时

在那些功能测试自动化完全由测试人员负责的项目中，一个重大挑战是打破测试人员和开发人员之间的无形隔阂。在这种情况下，已经不需要证明测试自动化的价值，也不需要排除测试环境的问题，但是团队必须变得更加具有协作意识。

这个问题更多是文化上的（之后会详细描述），但有时候也跟财务相关。如果使用如QTP这般昂贵的测试自动化框架，按人数购买许可，那么开发人员和业务分析师就会被特意隔离，不让他们接触测试。一旦团队改变态度倾向于协作，那么他们就能够在需求说明上进行协作，进行验证自动化时也不用去修改需求说明。

有些团队碰到某个问题却无法使用现有的自动化工具去做适当的测试，于是他们就只能朝可执行需求说明的方向上努力了。正是这种状况让他们获得了一个很好的理由来开始使用支持可执行需求说明的自动化工具。(详见附录中“工具”小节的例子，其他关于工具的文章请见http://specificationbyexample.com)

→这些团队发现，使用支持可执行需求说明的自动化工具后，开发人员会更多地参与到测试自动化中，同时商业用户也能够对测试有更深入的了解。

如此一来，开发人员会变得更加乐意参与到测试自动化中，并开始在自己的机器上运行这些测试，因为他们看到了从功能测试上获得快速反馈的价值。商业用户也能理解使用可执行需求说明的工具进行自动化的测试，并会参与制定相关的验收标准。在这之后，转移到设计可执行

需求说明并事先进行测试的过程，就相对容易了。

当Rob Park的团队与一家大型保险公司合作时，系统需要生成PDF格式的保险证明，他们以此为由，引入了一个自动化可执行需求说明的工具，并将功能测试转移到了开发周期的早期阶段。Park说：

“QTP无法对它进行测试——它能够验证弹出窗口没有出现错误信息，仅此而已。我想要的是能够让开发人员先在他们的机器上运行测试，但这确实是**QTP**的限制之一（由于需要按人数购买许可）。后来我使用了**JBehave**。我们几乎是一次性彻底地抛弃了**QTP**的一切，实际上这只花了一周的时间。现在我们能够让这些验收测试来驱动底层控制器的设计了。**”**

在Weyerhaeuser公司，Pierre Veragen和团队使用了自定义的测试自动化工具，它通过录制对用户界面的操作来进行测试。维护成本很高。有一次，某个变更导致了很多测试出错，他评估后发现，使用新的工具重写现有测试比重新录制出错的测试所花的时间要少，于是他提出采用FitNesse。采用FitNesse使得团队能够与工程师一起更紧密地协作，设计可执行的需求说明，也坚定了他们采用实例化需求说明的决心。

4.1.5 使用测试驱动开发作为踏脚石

适用于：开发人员对**TDD**有较深认识的时候

→引入实例化需求说明的另一个常见策略，就是从（单元）测试驱动开发上入手，特别是在开发新项目的时候。

相对于实例化需求说明，测试驱动开发的实践在业界更具知名度并且有更多的文档。如果一个团队已经把TDD实践运用得很好，那么很可能不需要再给他们展示自动化测试的价值了，也不需要为了让软件更具可测性而修改设计。可执行的需求说明可以看作是测试驱动开发对业务规则的扩展。（验收测试驱动开发常常是实例化需求说明的同义词。）

在ePlan Services公司，当他们第一次实施实例化需求说明时，Lisa Crispin使用了下面的方法：

“一开始，我无法让人们对验收测试感兴趣。按照**Mike Cohn**的建议，我就选了一个故事，去找了正忙于此故事的开发人员并问他：‘我们可以针对这个故事结对编写一个测试吗？’开发人员将会看到这是件多么简单的事。在下一轮迭代里，我选择了不同的故事和不同的开发人员。在他没有真正理解需求的地方，我们很快就发现了一个缺陷。这样开发人员立刻就看到了它的价值。**”**

当团队对TDD有较深的认识时，很容易解释可执行的需求说明：可执行的需求说明就是针对业务功能的测试。

4.2 如何开始改变团队文化

从很大程度上来讲，实施实例化需求说明需要进行文化变革——让大家在处理需求时进行协作，并改变业务人员、开发人员以及测试人员参与制定需求说明的方式。下面是一些有用的想法，有助于改变团队的文化。

4.2.1 避免使用“敏捷”术语

适用于：在一个抵制变化的环境中工作时

敏捷软件开发的方法饱受术语和流行语的困扰。Scrum 、立会、用户故事、功能清单(backlog)、大师(master)、结对编程，以及其他一些诸如此类的术语，很容易让人产生误解并导致混乱。对有些人而言，它们甚至会喧宾夺主，让人提心吊胆。术语造成的焦虑，是导致大家回退到从前并抵制任何过程变更——或者被动地等待失败到来的一大原因。以我的经历来看，许多业务人员很难理解开发团队使用的技术术语，因而很难理解关于过程改进的想法，很难去配合团队完成过程改进。

→无需使用技术术语就能实施实例化需求说明，这是完全可能的。如果工作环境抵制变革，那么开始变革时就一定要避免使用术语。

不要提及用户故事、验收测试或可执行的需求说明——实施实例化需求说明的时候不要提供定义。这样反对你的人就很难找到什么理由来反对。把实例化需求说明解释成一种为澄清需求而搜集实例、设计测试，并将它们自动化的过程。其他事情就让大家自己去发现吧。

在RainStor公司，Adam Knight没费多大力气就实施了实例化需求说明的大部分关键元素。并没有什么前期的规划，整个过程就那样慢慢推进着，Knight说公司其他人都不知道什么是实例化需求说明。他说：“实际上大家并没有意识到有什么特殊的。”对他的团队来说，那只是一个他们自己建立起来的过程而已。

Pierre Veragen使用类似的方法帮助Weyerhaeuser的一个团队改进了他们的软件过程。那个团队维护着一个遗留系统，有上百万行代码。他们抵制实施任何以敏捷为名的东西。Veragen并没有说什么豪言壮语，他只是建议在用户界面之下进行自动化测试，以便让测试更加高效。当大家接受这种做法后，他接着建议开发人员在他们自己的机器上运行测试，以便获得更快的反馈，让测试和开发齐头并进。Veragen就这样带着大家一起做，并密切注视着团队的动向，最终他改变了团队成员的看法：测试并不是开发完成后才做的事情。这样的变化花费了大约6个月的时间，主要是因为自动化测试套件必须达到一定的规模后，开发人员才能在代码中引入问题后看到有测试没有通过。Veragen是这么评论的：

“工程师意识到，在他们的机器上执行失败的测试实际上指出了代码中存在的问题。当开发人员碰到这种情况时，他们就明白了，不会再询问为什么他们必须要运行那些测试。”

实施过程变更无需技术术语，只需使问题变得显而易见，同时逐渐把大家推向解决问题的正确方向上就可以了。当团队想出某个问题的解决方案时（即使是在一些帮助下想出来的），他们会产生一种主人翁意识，也会更自觉地去遵循过程变更。

4.2.2 确保你得到管理层的支持

实施实例化需求说明的时候，大部分团队要显著地改变他们的工作方式。对很多团队而言，这意味着不仅要改变他们处理需求说明、开发以及测试的方式，而且要学习如何与团队内部以及外部的项目干系人进行更好的协作。

很多人会对角色转变感到困惑。测试人员必须更多地参与分析，开发人员必须更多地参与测试，分析师必须改变他们搜集和沟通需求的方式，业务人员则必须更加活跃地准备需求说明。如此巨大的改变需要得到管理层的支持，否则，它们注定会失败。Clare McLennan说道：

“项目的成功需要得到管理层的支持，尤其是当已经有一个系统的时候，因为要使之达到运行良好、稳定的程度，必定需要花费相当多的时间。你必须参与所有的迭代，查看哪里尚未稳定、哪里出现奇怪

的响应，修正它们，并不断重复。经过大约一年的时间，你才会得到一个非常宝贵的系统。反之，如果不那么做，或者你认为手工进行用户界面测试能很快修正缺陷，那么你最终得到的系统将会是一个难以维护并且价值不大的东西。”

刚开始的时候，自动化可执行的需求说明对许多团队来说都具有挑战性，因为从概念上来说，它与测试人员和开发人员所熟知的自动化测试是不同的（详见第9章）。团队必须学习如何使用新的工具，找到一个设计可执行需求说明的好方法，并组织好他们的活文档。在起初的几个月里，开发团队的生产率在升高以前，必定会先有所下降。这也需要管理层的理解、批准和支持。

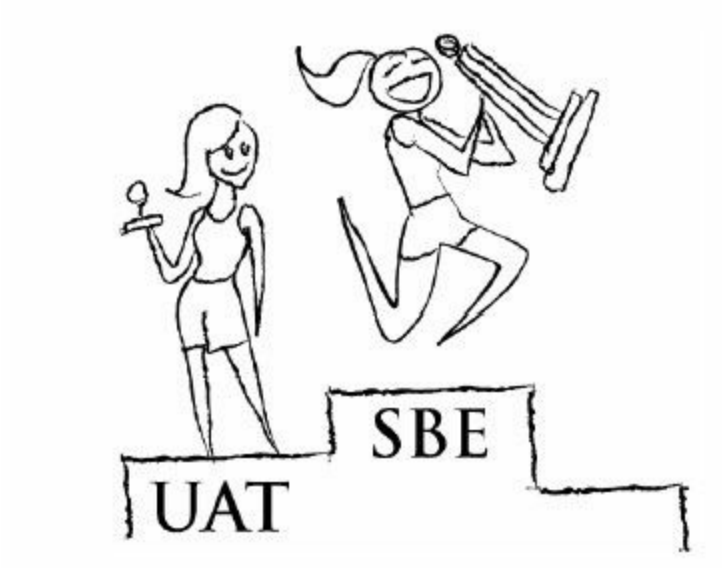
→没有管理层的认可和支持，过程变更成功的几率很小。

如果管理层不予支持，反而施加压力，那么大家就会退回到做事的老路上去，并开始保护他们自己的位置，而不是进行协作。与管理层分享第1章列出的成功故事和实例化需求说明的好处，应该有助于获取他们的支持，但如果失败了，那最好还是先不那么雄心勃勃，而以较小的步骤去改进过程。

4.2.3 把实例化需求说明当作是比执行验收测试更好的方式来推销

有些团队，包括那些在严格管理的环境中工作的团队，认为把用户验收测试作为软件交付中的一个阶段是没有必要的。（有些公司把这个阶段称为用户验收测试或业务验收测试。）虽然这并不代表他们不为用户验收做测试，但制定并检查验收条件和把用户验收测试作为软件交付的一个阶段是不同的。用户验收测试是非常重要的，不应该放到最后才去做。可执行的需求说明以及频繁的验证使开发团队持续不断地去检查用户的验收条件。只有所有的验收测试都通过，产品才会交付到用户手里。

如果可执行的需求说明足够广泛，而且验证足够频繁，那么开发团队与客户之间的信任会增强到这样的程度：在软件交付后，手工验证软件的功能变得不再必要。（当然，这并不代表测试人员可以在交付前不进行探索性测试。）



→我认为，大部分团队是能够以避免把验收测试拖到最后为理由证明实施实例化需求说明的花费是值得的。改变过程，让团队能更快地完成目标，会带来可衡量的经济利益，这样也就可以证明在过程变革中的投资是值得的了。

短周期的迭代或者基于流程的开发显著提高了潜在发布的频率。比方说，你想在接下来的12个月里，发布12个版本（大部分我采访的团队会发布2倍于这个数字的版本），用户验收测试平均需要3天的时间。这意味着，在未来的1年里，你要花36天的时间做用户验收测试，这还是从最佳情况考虑：没有发现任何问题，软件总是被客户所接受。（但如果这样，为什么还要花3天的时间去测试呢？）实际中更常遇到的情况是：验收测试放到最后才去做，然后返工，然后一年中至少再花上两个月的时间重新进行测试。

如果一开始就与其他人协作确定验收条件，并实现自动化验证，那么你就不必浪费时间去手工测试及返工了。自动化有一定的成本，但实例化需求说明可以显著减少产品上市所需的时间。

实例化需求说明还有很多其他好处，但这一点是最容易展现给商业项目干系人看的，也是最容易量化的。如果你要说服商业项目干系人接受这种过程变更，不妨试着把它能使产品每年提前两个月投放市场作为卖点。

4.2.4 不要让测试自动化成为最终的目标

在我采访的团队中，最常见的早期问题之一是他们把功能测试自动化当成了过程变更的最终目标。业务人员通常会认为功能测试自动化是测试相关的工作，因此他们没必要参与进去。开发人员需要理解，为了

改善沟通，自动化测试应该是人们可以读懂的，否则他们在对测试进行自动化的时候，只会考虑最大限度地减少开发的工作量。

→团队只关注测试自动化时，就不会更好地协作。

这种方法会导致测试过于技术化，使得测试只是一些脚本，而不是需求说明，这是一种常见的失败模式（详见8.3.2节）。从长远来看，这样的自动化测试会成为过程变更的障碍，而不是催化剂。

如果你把功能测试自动化当成实例化需求说明的一个步骤，那就要让团队的所有人都清楚最终的目标是什么。当功能测试自动化站住了脚，就该进行下一步行动了。

4.2.5 不要太关注工具

我采访的人员中，有3位一开始只是想选择一个好的工具。有些开发人员听说过FitNesse或者Cucumber，然后他们决定在项目中试试看。这种错误我自己也犯过，但这种做法成功的机会不大。

→实例化需求说明并不以程序员为中心，而程序员独自使用一个工具不会取得很好的效果。

这种方法往往会导致这样的局面：程序员想实现可执行的需求说明，但却使用了非技术性的工具，试图管理技术性的、面向开发人员的测试。这无疑是在浪费时间。

在开发人员关注于特定工具的3个案例中，只有Ian cooper的团队成

功建立了一个良好的过程，他们效力于Beazley公司。他们很努力地让测试人员和业务分析师参与进去，接着调整了他们编写和组织测试的方式。同时，他们对工具带来的好处精益求精，并寻找更简单的方法来获取那些好处。

在另外两个案例中，团队过分关注于工具，而不是去关注高层次的协作和过程变更。最终，他们浪费了很多时间和精力，建立的一套技术性测试方法让业务人员和测试人员根本没法使用。他们在测试维护上花费了非常多的精力和时间，却没有从实例化需求说明中得到任何好处。

4.2.6 在迁移过程中，遗留脚本也要有人维护

适用于：在遗留系统中引入功能测试自动化时

使用新的工具去重写功能测试并将它们自动化需要一定的时间。在新的验证系统成长到一定规模前，现有的测试应该予以维护，并使其保持更新。解决这个问题的一个好方法是：在做近期计划时，委托一个人专门去维护并更新老的测试。

James Shore和Shane Warden在《敏捷开发的艺术》一书中，把遗留项目的过程变更描述成“蝙蝠侠”模式。蝙蝠侠是一个专门解决紧急问题、修正重大缺陷的人，而团队的其他成员则继续新功能的开发。Markus Gärtner采用过这种方法，逐步地把一套测试迁移到为可执行需求说明而设计的自动化工具上。他详细介绍了他的经验：

注释：①James Shore和ShaneWarden，The Art of Agile

Development(O'ReillyMedia，2007)。

“当我们从基于**shell**脚本的测试过渡到基于**FitNesse**的测试时，起初只有两位成员专注于新的东西上，而维护遗留测试脚本的人则有**3**个。随着时间的推移，我们有越来越多的测试人员参与到新的方法中。先是**3**位，然后又加**1**位。最后，我们可以彻底把老的脚本抛弃掉。

这背后的思想就是**‘**蝙蝠侠**’**模式，他只管解决问题。我记得有些同事甚至从**Hot Wheels**买了一辆玩具车——蝙蝠车，并将它交给了我们当时的**‘**蝙蝠侠**’**。最初的时候，我就有过让大家轮流当蝙蝠侠的想法，但从来就没尝试过，因为当时我的同事都不喜欢分享知识。现在我们采用了这种新的方法，我试着让大家轮换担任蝙蝠侠的角色，这样在过渡过程中，每个人都可以接触到旧的东西和新的东西。让每个人都认同新的做法是至关重要的。**”**

→通过委派一个专门的人员来更新遗留事项，团队就可以更快地朝着迁移到新过程的目标前进。

这个想法与Alistair Cockburn的“牺牲一人”策略很像，留一个人专门处理让人分心的任务，其余成员全速前进。

注释：②http://alistair.cockburn.us/Sacrifice+one+perSon+strategy

4.2.7 跟踪哪些人在运行（以及没有运行）测试自动检查程序

适用于：开发人员都不愿意参与时

当开发人员有很强的结构化过程背景（程序员编写代码，测试人员对其进行测试）时，团队就难以让程序员也参与到过程中。为了让实例化需求说明产生作用，这一点必须要改变。

Pierre Veragen有一个独特的解决方案，可以让程序员参与进来。他创建了一个简单的集中化的报告系统，告诉他可执行的需求说明在何时何地被检查过：

“在**Fixture**代码中，我放入了一点小东西，它告诉我人们什么时候在他们的机器上运行了测试。那个小组的成员都有点儿不善沟通。我用这种方式找出了人们什么时候没有运行测试，然后与他们进行交谈，以便了解发生了什么问题，看他们是否遇到了困难。这种做法是为了获取更加客观的反馈，而不是一句**‘**是的，它运行良好**’**。**”**

通过跟踪提交前谁没有执行测试，他能够集中精力去帮助那些遇到问题或者需要帮助的成员。Veragen说，由于所有程序员一开始就了解这个过程，因此他只跟踪某人是否执行了测试，而不是监视实际的测试

结果。

→通过监视测试是否执行，来让程序员执行自动检查程序。

在较大的团队里，教练无法总是与所有成员一起工作，因此这是一种有趣的方法。我估计这种方法的效果与公布高速公路上高速摄像机的位置有点类似——程序员知道有人在盯着，因而他们运行检查程序时会更加仔细。

4.3 团队如何在流程和迭代中集成协作

当团队开始实施实例化需求说明时，其中一个最大的挑战是理解如何将协作融入到交付周期中。

实例化需求说明和瀑布式分析的区别

我在大会上遇到的很多人都误以为增量地建立一个文档系统意味着回到瀑布式的思想，需要做大量的前期分析。**2009**年**11**月，**Dan North**在他的演讲**"**如何将**BDD**推销给业务人员**"**中说，**BDD**实际上是压缩到两周的**V**模型。虽然这种描述并不完全准确，但它是一个良好的开端。

注释：①http://skillsmatter.com/podcast/agile-testing/how-to-sell-bdd-to-the-business

瀑布式分析的方法与实例化需求说明试图实现的东西有一些本质区别。理解这些基本原则很重要，因为它们有助于你把这些实践融入到自己的过程中，无论你的过程是什么样子的。以下这些是实例化需求说明区别于规划性分析方式的关键因素。

通过快速周转来提供快速反馈和重点；高效地完成软件的一小块，而不是试图一次性处理一大块。

强调有效、高效的沟通，而不是冗长、乏味的文档。

建立共享所有权，这样在需求说明变成代码或测试的过程中，开发人员与测试人员不会互不通气。

整合跨职能团队，为了制定正确的系统需求，测试人员、分析师和开发人员一起进行工作，而非各自为战。

过程变更没有通用的解决方案，每个团队都需要决定如何最好地扩展他们交付软件的方式。接下来我介绍一些具有代表性的例子，帮助你开始着手变革。我挑选了3个很好的案例研究，每一个都代表一种受欢迎的过程。Global Talent Management团队采用基于过程的看板框架，Sierra团队使用基于迭代的极限编程过程来交付软件，而Sky Network Services集团基于Scrum进行迭代。

4.3.1 Ultimate软件公司的Global Talent Management团队

Ultimate软件公司有一个人力资源管理系统，该系统拥有16组团队，Global Talent Management团队是其中之一。该团队由一名产品负责人、一位用户体验专家、4名测试人员和10名开发人员组成。当我采访该团队的Scott Berger和Maykel Suarez时，他们的项目已经开展了8个月。那个团队使用的是看板的工作流过程。

由于产品分析师（兼作分析师和产品负责人）很忙，团队在与他协作制定需求说明时，试图更有效率地利用他的时间。产品分析师使用“故事点”在更宏观的层次上解释一个故事（在他们这个案例中，故事点指的不是复杂度的估计，而是解释故事的列表项）。编写故事点时，他们尽量避免使用技术专用语言。然后该故事就被添加到看板上当作功能清单的一部分。

每日例会限制在30分钟之内，首席工程师、产品分析师以及团队里任何对功能清单感兴趣的人一同参加会议。他们快速地过一遍产品清单上的故事，检查每个故事是否被正确地分割、是否讲得通、是否可以由一个团队在4天之内完成——这是他们为每个故事设定的期限。在会议上，他们还会去理清没有解决的问题，盘点故事的依赖项。

之后，故事就进入了等待编写实例化需求说明的队列，它们也被用作验收测试。将要实现该故事的人与测试该故事的人一起结对，编写这些需求说明的概要。（他们在团队中没有正式的测试人员角色和开发人员角色，但是为了方便起见，在本小节里，我将使用这两个角色来指代结对的这两个人。）Berger解释说：

“通过这种结对，我们能够减少这个故事所需的测试，因为开发人员对代码更为熟悉。事实证明这种做法相当成功，因为在最初的故事审核中，可能会遗漏一些不一致的地方以及设计缺陷，而这种结对则可以有效消除这种问题。”

他们定义好大纲概要之后，测试人员通常以Given-When-Then的格式来完成各种场景。结对的两个人和产品分析师在一个深度“故事知识和信息传递”(SKIT)的会议上审核这些场景。用这种做法，万一分析师不在场，程序员和测试人员也可以写出良好的需求说明。产品分析师审核过这些场景后，团队就将它们当作需求。在故事交付前，除了少数的文字修正，不允许再对这些需求做变更。

然后，开发人员通常会在实现产品代码之前，自动化这些场景。这让测试人员可以有更多的时间去完成探索性测试。虽然测试人员可能也会去编写自动化场景，但这不再是他们的工作重点。Berger说这样的协作使得他们可以高效地工作：

“在自动化方面，熟悉代码的开发人员会更快，事实上他们编写的

自动化代码可以直接访问他们自己编写的对象（而不是通过用户界面来自动化），而且由于很多错误条件和组舍都明确描述了，所以他们在开发过程中可以获得更多益处。由于我们是在图形用户界面之下进行测试的，所以测试执行得更快更稳定。测试人员也可以花更多的时间来执行程序代码并提供反馈。”

在开发阶段结束前，所有的可执行需求说明都必须运行通过。在运行测试阶段会集成其他团队的工作，所有的测试将再次运行。而在等待审核阶段，团队要给产品分析师做一个快速的产品演示，让他验收。

据Berger说，这个过程产生了非常高质量的结果：

“通过和产品分析师的密切舍作，并将测试作为需求的基础，我们能够实现非常高的质量。业务人员采集了许多度量指标，其中一个指标我认为很好地说明了我们在提升质量方面的努力，那就是缺陷检测效率**(DDE，Defect Detection Efficiency)**。我们**Global Talent Management**团队的**DDE**是**99%**（**2010**年第**1**季度到第**3**季度）！”

4.3.2 BNP Paribas银行的Sierra团队

BNP Paribas银行的Sierra团队在开发一个后台参考数据管理和分配系统。团队由8名开发人员、2个业务分析师和1个项目经理组成。因为没有专门的测试人员，所以团队里每个人都需要负责测试。他们的项目干系人是不在现场的商业用户。变更需求通常需要大量的分析，要与项目干系人协作。

项目已经进行了大约5年的时间，所以已经相当成熟，并且业务分析师拥有很多现成的可执行需求说明，可以当作例子来使用。Andrew Jackman曾经是该团队的一员，当我采访他时，他说这是一个金融服务行业里的罕见例子，他们几乎完全遵照规范来应用极限编程。

他们的开发过程始于项目经理，项目经理事先为迭代挑选一些故事。在迭代开始之前，业务分析师同远程的项目干系人一起工作，准备详细的验收条件。他们使用现有需求说明的实例来驱动新需求说明的结构。如果新的需求说明与现有的任何一个需求说明都有显著的不同，那么结对的两个开发人员将对测试进行审核，以便提供早期的反馈并确定相关测试可以被自动化。

他们的迭代是两周一轮，始于第二周的礼拜三。当迭代开始时，整个团队聚在一起开计划会议，并按照优先级来审核本次迭代将要完成的故事。会议目的是为了保证所有的开发人员都能理解故事的内容，估算故事的大小，并检查技术依赖项，看是否可能进一步改善交付顺序。他们还会将故事分割成开发任务。当故事在计划会议里进行审核的时候，该故事的验收条件通常已经确定好了。

团队偶尔会发现某个故事不是很好理解。如果他们采用的迭代周期是一周，此类故事可能会破坏流程，但是两周长的迭代周期则可以让他们处理这种状况，而不会对整体流程带来多大的影响。

接着，开发人员以结对的方式来实现这些故事。在结对的两个开发人员完成一个故事并通过所有相关的测试后，业务分析师将花些时间去做探索性测试。如果分析师发现意外的行为，或者他意识到团队没有完全理解某个故事对系统带来的影响，那么他将使用相关的实例来扩展需求说明，并把该故事返回给开发人员。

4.3.3 天空网络服务部门

英国天空广播公司的天空网络服务部门在维护一个宽带的配置供应系统。该部门由6个团队组成，每个团队有五六个开发人员和一两个测试人员。整个部门共享6名业务分析师。由于团队单独维护不同成熟度的功能组件，所以每个团队的流程略有不同。

整个部门使用的是基于Scrum的过程，迭代周期是两周。在一轮迭代正式开始的前一周，他们会组织一个预先计划协调会议，每个团队抽出两三个人参加。该会议的目的是为故事排列优先顺序。当他们开会时，业务分析师已经为每个故事收集并指定了一些高层次的验收条件。在会议之后，测试人员将开始编写带实例的需求说明，这通常需要业务分析师的协作。在迭代正式开始之前，每个团队将至少拥有一两个故事，各自带有已经做好自动化准备的、详细的、带实例的需求说明。

迭代从第二周的周三开始，最初会有一个跨团队的计划会议，让每个人都了解整体的进度以及这轮迭代的商业目标。然后每个团队单独召开计划会议。某些团队只花15分钟简要地浏览一下所有的故事，而有些团队则要花上几个小时来研究细节。Rakesh Patel是该项目的一名开发人员，他说这主要取决于底层组件的成熟度：

“当我们接手的工作涉及一个存在很久的组件，而我们只是为它增加一些消息时，对整个团队来说，在选择该故事卡进行工作之前，没必要让每个人都知道它涉及什么内容。而有些团队正在开发一个新的图形用户界面，面对全新的功能，这时可能更舍适让整个团队坐下来深入讨论并规划需要完成的内容。在这个时候，我们可能还会讨论一些非功能性的需求、架构等。”

在计划会议之后，开发人员开始对那些已经有实例化需求说明的故事展开工作。业务分析师和测试人员针对当前迭代里计划的所有故事编写验收条件。一旦完成某个故事的需求说明，他们就与指定的“故事负责人”（详见后文）开会并仔细检查所有测试。如果每个人都觉得完成某个故事所需的信息已经足够，那就给那张故事卡贴上一张蓝色的贴

纸，表明那个故事可以进行开发了。

开发完成后，业务分析师将再次审核需求说明，并在卡片上贴一张红色的贴纸。然后，测试人员对故事运行一些额外的测试，当测试通过时就加上一张绿色的贴纸。有些故事在测试之后还需要支持团队、数据库管理员或系统管理员的审核。审核之后，数据库管理员给故事卡贴的是一个金星，系统管理员贴的是银星。贴纸可以确保所有涉及该故事的人员都对目前的状况一目了然。

SNS团队没有组织大型的需求说明会议，他们组织的是一个流过程。他们使用的需求说明制定过程依然有两个阶段：业务分析师和测试人员对所有实例进行前期准备，然后与开发人员一起进行审核。这让开发人员有更多的时间去关注开发工作。故事负责人在与其他开发人员结对的过程中负责有效地传递信息，而不需要让所有开发人员都参与审核来确保他们理解故事。

前面3个例子展示了团队如何将协作融入短迭代甚至基于流的过程中，证明了没有通用和全局适用的方法来将过程结构化。所有团队都成功地将协作集成到他们自己的短发布周期中，但是根据团队结构、商业用户的参与性以及变更进入交付流程的复杂性等的不同，他们使用的方法也不尽相同。

故事负责人**(Story Champion)**

SNS团队在轮换故事的结对开发人员时，采用了故事负责人来确保信息的有效传递。**Kumaran Sivapathasuntharam**是该项目的业务分析师，他说：

“故事会分配给某个特定的开发人员，他会坚持到该故事完成。这确保了每个故事都有一个联络人——这样当故事出现问题时，可以找该故事的负责人进行谈话。一个人从故事的开始坚持到结束，这样结对的两个人不会都被它所牵绊，他们可以不断改变结对的人员而仍然自始至终保持连续性。**”**

Ultimate软件公司的**Global Talent Management**团队有一个类似的角色，他们称之为故事担保人**(story sponsor)**。据**Maykel Suarez**所述，担保人负责与其他团队进行沟通、在看板上跟踪进度、在每日例会上检查状态以及清除障碍。

4.4 处理签收和可追溯性

对有些团队来说，敏捷项目没有文档或只有少量文档所带来的一个主要问题是需求的缺乏。这使得签收需求或交付变得很困难。整体上来看，软件开发行业远不如十年前关注签收。有些情况下，由于监管约束

或商业安排，签收还是需要的。

实例化需求说明提供了有关需求的工件：活文档。活文档可用于追溯，这使得敏捷过程可以应用于受管制的行业。Bas Vodde和Craig Larman在Practices for Scaling Lean and Agile一书中提到了美国核工业的第一个敏捷开发项目。那个项目的团队使用了可执行的需求说明以确保需求完全可追溯，这在核领域和其他对安全要求十分苛刻的领域是非常重要的。另一方面，由于使用了高动态的、迭代的协作方式来产生这些工件，前期签收几乎是不可能的。下面是一些如何处理签收和可追溯性约束的方法。

注释：①我也很想把这个案例研究加入到本书中，但不幸的是，我没能找到任何愿意和我谈谈这个项目的人。

4.4.1 在版本控制系统中保存可执行需求说明

→我访问过的一些人说，把可执行需求说明和产品代码放在同一个版本控制系统中，是成功实施过程的最重要的做法之一。

许多自动化工具支持纯文本文件的可执行需求说明，所以它们同样也支持版本控制系统。因此你可以很容易地对需求说明和源代码做标签(tag)和分支(branch)。这样你可以获得测试的最新以及正确版本，用以验证产品的不同版本。

版本控制系统有很高的可追溯性，因为它们让你可以瞬间找到什么人在什么时候出于什么原因修改了哪个文件。如果把可执行需求说明保存在版本控制系统中，那么将能轻易追溯到需求和需求说明。通过实例化需求说明，可执行需求说明将直接和程序代码联系在一起（通过自动化层），这意味着代码也具备可追溯性。

与那些保存在单独的需求或测试工具中的需求说明相比，版本控制系统中的可执行需求说明更不容易丢失。

4.4.2 通过导出的活文档来签收

适用于：逐个迭代签收

实例化需求说明应当有助于在项目出资人和交付团队之间建立信任，并免除签收的过程。如果确实因为一些商业或政治原因需要签收需求，那么你可以使用活文档系统。

→如果需要在实现功能前签收需求说明，并且可以在每个迭代里都这么做，那么你可以根据下一个迭代所计划的可执行需求文档创建一个**Word**或**PDF**文档，然后在上面签收。

有些自动化工具，比如Cucumber，支持直接导出为PDF，这可能对

签收过程有所帮助。

4.4.3 签收的是范围，而非需求说明

适用于：签收较长的里程碑

→如果你要签收的内容比一个迭代所能交付的要大，那就试着对范围而非对具体的需求说明进行签收。比方说，签收用户故事或用例。

Rob Park在为一家美国大型保险公司工作时使用了这个方法。他的团队签收时保持着瀑布式的审批过程，但显著地减少了需要签收的材料。Park解释道：

“在我们所能控制的事情之外还有一个更大的过程。业务分析师使用带模板的**Word**文档，但是他们把模板从**8**页减少到了**2**页。有一个故事卡审批过程，项目的出资人在只有业务分析师看过故事卡的情况下就签收了。所以他们只在公司的高层级里使用这个瀑布过程，而一旦进入到团队就是另一回事了。**”**

在这个案例中，使用Word文档纯粹是因为合同规定了一个故事进入开发前需要文书工作。在范围获得审批后，团队使用可执行需求说明作为他们唯一的需求来源。

4.4.4 在“精简的用例”上签收

适用于：受监管的签收需要详细的内容时

在严格的监管环境中对范围进行签收可能是不太现实的。来自Knowledgent Group的Mike Vogel曾在一个医药工业的项目中工作过，他们使用的过程是基于Scrum和XP的扩展，以便满足监管需求。他的团队使用用例，因为单独使用用户故事不能达到监管系统的标准。

团队使用精简的用例（他们称作“结构化的故事”），这样最初的捕获和持续演化就不是个大问题。那些用例避免使用关于数据和决策的大多数细节（这些被抽取到单独的数据部分）。Vogel解释了这个方法：

“在一个用例中，你需要为每一块数据起一个名字，客户会将它理解为领域语言的一部分。数据块的描述给出了结构和规则，其描述的是数据——而非实例。实例是在（验收）测试/实例化需求中，我们通过用例构建实例，这些实例覆盖并展示了所命名的数据块的所有可能变化。你根据他们提取出来的数据块变化实例。**”**

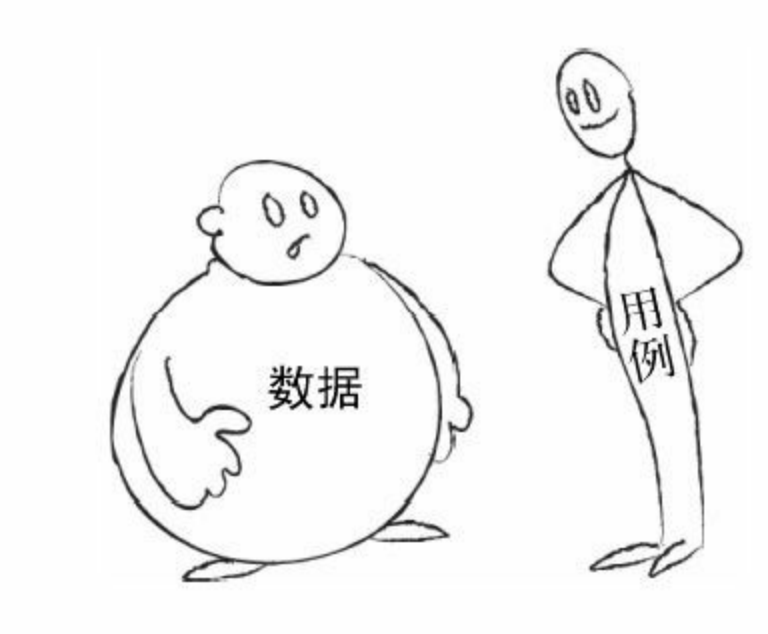
→对“更轻量的用例”做签收，而无需实例。

Vogel的团队用这些轻量的用例构建了需求文档，但不包含任何实例。其结果是一个大型项目的文档只有不到100页，按Vogel的说法，还“使用了所有监管要求的模板”。整个项目中，他们与客户协作，共同说明了用例和实例：

“我们和客户一起坐在团队房间里，尝试每次写出一个附带实例的

用例。讨论是围绕着具体的实例进行的。我们最终会加入一些细节，并且客户还会审查。”

团队通过这个方法可以在一些东西上完成签收，这些东西非常像传统的需求说明，但又不会太过细节。客户使用的列表是基于高层次的用例，而细节是后来在迭代和协作过程进一步充实的。



4.4.5 引入用例实现

适用于：签收时需要所有的细节时

Matthew Steer参与过一些基于结构化过程（统一软件开发过程）的项目。该过程要求对所有的细节进行签收，需求说明被捕获成用例。另外，Steer和他的团队还引入了用例实现，它通过实例有效地演示了用例。这使他们能够在结构化过程中使用实例化需求说明。Steer说：

“需求被捕获成用例和额外的非功能性需求的需求说明，用法非常传统，照章办事地捕获。通过用例，我们生成了用例实现、实例以及用例执行的场景。我们用很多参数创建表格并与数据相关联，而后通过流程图来显示如何实现用例。通过使用真实的场景，用例实现就变成了商业意图的一个可用的版本。”

→在方法论雷达监控之下，添加诸如用例实现的细节是在正规过程中引入实例化需求说明的一个不错的想法。当商业舍同需要对需求进行签收但还允许之后对细节进行变更时，这种做法同样有助于实现实例化需求说明的概念。

Steer的团队和之前提到的其他团队一样，使用实例（即便它伪装成

用例实现）而不是只使用用例或针对更通用的需求制定的测试。这使得他们的交付过程更加有效。

从技术上来说，活文档系统能立刻提供需求变更的可追溯性，因为团队使用版本控制系统保存可执行的需求说明。迭代开发和预先签收通常是冲突的，但是当过程进行了变更并且交付团队受到商业用户信任的时候，你可以使用本节的技巧来应对这一冲突。活文档系统提供的透明度以及各方协作制定需求说明，都有助于消除签收的必要性。

4.5 警告信号

你可以跟踪自己的进度，检查实例化需求说明实施得是否恰当。对于任何指标，要确保指标本身不要成为目标，否则的话，你可能会对过程进行局部优化以便达到某个指标，而不顾这会伤害长期的结果。使用指标作为度量标准，检查是否需要对过程作出调整。

4.5.1 注意频繁改动的测试

在2009年的极限编程日上，Mark Striebeck谈到了Google为了推进他们的测试实践所做的事情。让我印象深刻的一个观点是，他们如何衡量某个（单元）测试的良莠。当某个测试失败时，他们会跟踪源代码的改动，直到测试重新通过。如果后台的代码更改了，他们会认为那个测试是一个良好的测试。如果测试更新了而源代码没有更新，他们会认为那是一个不好的测试。通过搜集这些统计数据，他们希望分析出单元测试的模式，确定什么因素会影响一个测试的好坏。

注释：①http://gojko.net/2009/12/07/improving-testing-practices-at-google

我相信相同的标准同样适用于可执行的需求说明。如果某个验证失败，你更改了代码，那就说明你发现并修正了一个问题。如果某个验证失败，而你不得不更改需求说明，那就代表需求说明编写得不太恰当。

业务规则应该比实现它们的技术要稳定得多。要注意频繁更改的可执行需求说明，我们要寻找好的方法将它们编写得更好。

你也可以统计你的团队为让问题保持在可控的范围内而花费在重构需求说明以及相关自动化代码上的时间。如果你在迭代中要花费很多时间来做这件事情，那就去寻找更好的方法来做自动化测试吧（请参考第9章，那里有一些很好的提示）。

4.5.2 当心回退

检查是否做错了什么，有另外一个很好的指标，那就是检查回退的出现。回退是指一个故事或者产品功能清单中的项目，在发布后不到一个月的时间内又返回到项目过程中。团队认为已经完成了，但事实上仍需要返工。不过，将来出于业务需要对产品进行创新、演进，而对现有的需求进行扩展，这就不属于回退。

一旦实施实例化需求说明，回退出现的次数应当会显著减少，直至鲜有发生。协作制定需求说明以及测试与开发更好的结合，应该可以消除由于误解造成的返工。回顾几个月回退的趋势，可以让你了解你们改善了多少。如果比率不降，那就说明实施过程的方法有些问题。

跟踪回退不需要很多时间，通常每轮迭代花上几分钟就可以了，但当遇到了挑战，或者需要证明实例化需求说明执行良好时，它都是非常有帮助的。在规模较大的公司里，它也可以提供令人信服的证据，表明实例化需求说明是很值得其他团队去实施的。对于更复杂的统计，也可以跟踪花费在回退问题上的时间，因为这个数字直接体现出浪费在开发或者测试上的时间和金钱。如果大家抱怨花费在自动化可执行需求说明上的时间是不必要的开销，那就比较一下几个月前他们花费在回退上的时间。据此建立起一个实施实例化需求说明的成功案例，应该是绰绰有余的。

一旦回退的数量有所下降，并且它们相对较少出现，那么就不必再跟踪了。如果遇到了回退，尝试了解它由何而来。我的一个客户有很多来自他们财务部门的回退。这表明他们与公司财务部门的沟通有问题，为此，他们寻求更好的方法让那个部门参与进来。

跟踪回退也是一个很好的方法，可以为引入实例化需求说明提供业务上的支持。它可以帮助团队查明由需求模糊以及需求说明里的功能分歧造成的浪费。

4.5.3 注意组织级的失调

很多团队在刚开始实施实例化需求说明的时候，是把它作为一种帮助他们在迭代中更好地协调活动的方法。而一旦熟悉了可执行需求说明，并且自动化代码变得稳定之后，应该可以在同一个迭代中完成一个故事，并彻底做好测试工作（包括人工的探索性测试）。如果测试人员滞后于开发，那就做错了。分析没有协调一致也是一个类似的警告信号。有些团队在相关迭代开始前就开始做分析，而他们仍然拥有规律性的间隔和流程。事先分析过头、分析不会马上被实现的东西，或者需要分析细节时却滞后了，这些都是过程出现问题的警告信号。

4.5.4 当心“以防万一”的代码

在Lean Software Development一书中，Mary和Tom Poppendieck写

道，软件开发中最大的浪费源是写了很多“以防万一”的代码——实际上不需要的代码。我不确定这是不是最大的浪费源，但我确实看到过大量的金钱、时间和精力浪费在没人需要的东西上。实例化需求说明可以显著减少这种问题，因为它会帮助我们建立共识，让我们知道要交付什么。Jodie Parker说关于需求说明的交流和协作帮助她的团队实现了这一点：

“当开发人员拿到一张故事卡时，他们会非常希望交付其中的一切，技术上追求尽善尽美，即便他们的指导原则是‘做最少的事情，获取最大的价值’。但效率要紧，而且我们总能在今后重新审视这些故事并加以改进。对此的一个解决方法是，通过交流沟通，以及持续不断地检查，从我们当前所做的当中是否可以看出清晰的商业模式。通过领域建模，你可以很容易地分解出任务。那些任务是你唯一需要做的。由于任务都很小，你可以按计划完成它们，但如果你没有，团队的其余成员轻而易举就会发现，他们会坦率地说出来。当有人在某个任务上已经工作了几天，那我们会在站立会议上与他进行交流。”

有些开发人员实现的功能会超出实例中的商定和规定，要注意这些成员。另一个可以避免“以防万一”代码的好方法是不仅明确想要交付的范围，同时也讨论清楚哪些不在交付范围内。

4.5.5 注意霰弹式修改

霰弹式修改是一个典型的编程反模式（也叫作代码异味）：对一个类做很小的改动后，你又需要对几个相关的类作出一系列改动。这个信号同样适用于活文档：如果对生产代码做了某个改动后，发现还需要修改很多可执行需求说明，那说明你有地方做得不对。组织好你的活文档，这样对代码进行一个小改动时，只需要对测试做一个较小的更改即可（请参考11.4节，其中有一些好的建议会教你怎么做）。这是降低自动化长期维护成本的一个关键步骤。

4.6 铭记

若要为开发团队提供及时的需求说明，实例化需求说明是一个不错的方法，因而它是在短迭代或基于流程的开发过程里取得成功的重要因素。

高效地完成软件的一小块，确保快速的周转和快速的反馈。

强调有效的、高效的沟通，而非冗长的、乏味的文档。

整合跨功能的团队，测试人员、分析师以及开发人员为了给系统建立正确的需求说明而一起工作。

事先为自动化的开销做好计划。

Part 2 第二部分 关键过程模式户故事描述了用户故事描述了用

本部分内容
第5章 从目标中获取范围
第6章 通过协作制定需求说明
第7章 举例说明
第8章 提炼需求说明
第9章 自动化验证而不修改需求说明
第10章 频繁验证
第11章 演化出文档系统

第5章 从目标中获取范围

F-16战隼战斗机可以说是有史以来最成功的喷气式战斗机。它的成功令人称奇，因为它成功战胜了一切困难。F-16战机设计于20世纪70年代，那时的喷气式战斗机追求的是速度，而打击范围、武器装备以及机动性在当时并不怎么受重视①。然而正是F-16出色的打击范围和机动性，使它很适合作战，这确保了它的成功。

注释：①请参考Kev Darling的书F-16 Fighting Falcon(Combat Legend)[F-16战隼战斗机（战斗传奇）](Crowood Press，2005)。

在《软件架构师必须知道的97件事》一书中，Einar Landre引用了F-16首席设计师Harry Hillaker的话，他说飞机最初的需求是飞行速度要达到2～2.5马赫。Hillaker询问美国空军这为何如此重要，得到的答复是“飞机必须能从战斗中逃脱”。尽管Hillaker的设计没有超过2马赫，但它让飞行员可以非常敏捷地从战斗中逃脱。它集很多创新于一身，包括无框气泡式座舱盖，可以获取更好的视野；倾斜的座位，可以降低重力对飞行员的影响；一个可以在飞行员前面投射作战信息却不会妨碍视线的显示器；侧装式控制杆，能够在高速飞行时提高机动性。有了这些功能，F-16完胜其他设计，而且生产成本更低。它赢得了设计比赛。30多年后，它仍然在生产。生产数量超过4400架，销往了25个国家②，这个型号取得了巨大的商业成功。同时，它也是最流行的战斗机之一，经常在动作电影中出现，比如《X战警2》和《变形金刚：卷土重来》。

注释：②请参考http://www.lockheedmartin.com/products/f16。

F-16是成功的，因为它的设计比用户要求的解决方案更好、更便宜。原先的那些需求，包括2.5马赫的速度要求，看似问题的一种解决方案，却并未有效地传达真正的需求。设计师没有直接去实现那些需求，而是去寻求对问题更深入的理解。一旦有了更深的理解，他们就可以找出真正的目标，并从那些目标中形成他们的设计，而不是从建议的解决方案或有关功能的随意期望中获取设计。这是成功产品设计的本质，在飞机研发中如此，在软件设计中也同样重要。

与我一起共事过的商业用户和客户，大多喜欢把需求描述成解决方案；他们很少会去讨论想要达到的目标，或者亟待解决的问题具有什么特殊性质。我见过太多的团队有一种危险的误解，他们认为客户总是正确的，客户要求的东西总是一成不变的。这导致很多团队盲目地接受客户建议的解决方案，然后竭尽全力去实现。成功的团队不会那么做。

成功的团队会像F-16的设计师们一样，先推开那些需求，获取更多实际问题的信息，然后进行协作，设计出方案。他们对待范围也是如此，范围隐含着解决方案，成功的团队不会把定义范围的责任推到其他人身上，他们会积极协作，同商业用户一起，确定良好的范围，以期达到他们的目标。这就是从目标中获取范围的本质。

通过协作从目标中获取范围无疑是本书最具争议的话题。在过去的5年中，软件开发价值链的概念迅速普及，提高了大家的认识，那就是要协作确定软件的范围，并从商业目标中获取范围。另一方面，与我共事过的大多数团队仍然认为项目范围不在他们的掌控之中，他们期望客户或者商业用户可以完全确定好范围。在做调研的时候，我发现了一种团队协作从目标中获取项目范围的模式，不过相比其他的关键模式，这种实践不太普遍。

本来我想舍弃这一章。但我还是决定把它编进本书，原因有以下3个。

在构建正确软件产品的过程中，确定范围扮演着重要的角色。没有正确的范围，其余的工作只是在做无用功。

未来，这会成为软件开发中最重要的一个课题，我想提高大家对于这一点的认识。

确定范围很适合基于价值链的设计过程，价值链的概念由于精益软件开发的普及而越来越受到欢迎。

在下面两个小节中，我会介绍一些确定范围的方法，无论对那些可以直接控制范围的团队，还是无法直接控制范围的团队，这些方法都是有效的。那些对项目范围有很高控制权的团队，可以马上积极主动地开

始构建正确的范围。遗憾的是，与我共事过的几个大型组织中的许多团队都没有这样的权利，但这并不代表他们不能影响项目范围。

5.1 构建正确的范围

用例、用户故事或产品功能清单中的条目提供了项目范围的广义定义。许多团队认为这些工件是商业用户、产品负责人或客户的责任。要求商业用户提供范围，实际上是依赖没有软件设计经验的人来提供高层次的解决方案。设计解决方案是最具挑战性也是最重要的步骤中的一个。引用Fred Brooks在《人月神话》一书中的话：“构建软件系统最难的部分是精确地定义构建的是什么。”Albert Einstein也说过：“问题的表述常常比它的解决方案更重要。”

当前，用户故事是敏捷和精益项目中定义范围最流行的方式。用户故事确实能很好地提高人们对软件项目中商业价值的意识。用户故事让我们可以和商业用户讨论他们可以理解的事情，让他们合理地安排事情的优先级，而不是要求他们在开发一个集成平台和创建事务CRUD（创建、读取、更新、删除）界面之间做出选择。需要注意的是每个故事应该拥有一个明确关联的商业价值。商业用户常常随意地选择价值的表达方式（它通常是冰山一角）。但是当我们知道故事应当交付什么的时候，就可以更深入地调查研究并提出替代解决方案。TechTalk的Christian Hassa解释说：

“人们告诉你他们自己认为需要什么，通过问他们‘为什么’，你可以找到背后的目标。许多组织不能明确地指出他们的商业目标。然而，一旦你获得了目标，就应该再反过来从已确定的目标上获取范围，可能你会丢弃掉原先假定出来的范围。**”**

这就是我在Bridging the communication Gap一书中称作挑战需求的做法的本质。我仍然认为挑战需求是一项重要的实践，但是这么做不够主动。虽然它确实比被动要好（我见过的大多数团队在确定范围时都是被动的），但还有许多新兴的方法和实践可以帮助团队更主动地获得商业目标。我们可以一开始就和商业用户一起制定出正确的故事，而不是事后对错误的故事作出反应。关键在于不要从用户故事开始，而是从商业目标着手并通过协作从目标获得范围。

5.1.1 理解“为什么”和“谁”

用户故事一般有3个部分：“作为……，我想要……，为了……。”还有一些其他的格式，但都包含这3个部分。

→要评估一个建议的解决方案，理解为什么需要某些东西以及谁需要它是至关重要的。

这些问题同样适用于更高层次的项目范围。事实上，在一个更高的层面上回答这些问题可以将项目推向完全不同的方向。

来自比利时iLean的Peter Janssens曾经作为甲方参与过一个项目——他是以解决方案的形式提供需求的人。他负责一个存储本地交通标志信息的应用程序。一开始对于比利时的数据他们只使用了简单的Access数据库，但是该应用很快就需要覆盖到世界上大多数国家。公司在每个国家都有一个数据收集器，他们都使用本地Access数据库，偶尔需要将它们合并在一起。

为了使工作更加高效并避免数据合并的问题，他们决定采用在线数据库并且使用一个网页应用来维护它。他们花了4个月联系供应商，比较报价，最终选定了一家供应商。这个应用的预计成本是10万欧元。但是就在他们认真考虑了谁需要使用这个应用程序以及为什么之后，项目发生了很大的转变。Janssens说：

“在决定是否做这个项目的前一天，开发部门的一个家伙为了更好地理解，再次询问了我当前面临的问题。我说：‘我们需要一个中心数据库的网页解决方案。’他说：‘不，不，我们不要直接跳到结论。不要马上描述你需要哪个解决方案，请先解释一下。’我又解释了一遍。然后他说：‘那么，你的问题实际上是需要在一个单独的数据源上工作，因为你不想浪费时间做舍并。’‘是的，’我说，‘正确。’

他提了第二个问题：‘谁会使用它？’我说：‘瞧，此时此刻我们有**10**个国家的数据，所以有**10**个人。’我们看了看数据库，意识到这类交通信息不会经常改变，可能每个国家一年就改变一两次。然后他说：**‘Peter**，听着，你的问题明天就可以解决。’第二天他在他们的**Citrix**（远程桌面）服务器上添加了数据库。**”**

应用程序总共需要支持10个用户，他们只在更新交通标志信息时使用它，而这种改动很少发生。Access应用程序足以应付这个数据量，他们真正的问题只有合并。一旦技术工程师理解了潜在问题，他可以提供比原来建议的解决方案要便宜得多的方案。Janssens解释说：

“我了解到这是一个真正的对峙局面——理解导致需求的核心问题总是很重要。所以理解‘为什么’很重要。最后，当我们谈到‘谁’的问题时，他想到了**Citrix**解决方案。通常一个月只有一个用户使用它。**”**

就算是在范围的层面上，也可能隐含了解决方案。还未确定可能的用户故事或用例，也没有为了划分任务而讨论需求说明，就有人建议使用网页应用程序，这个事实就隐含了一个解决方案。他们使用一个零成本的快速修复方案就解决了问题，而不是花5个月选择供应商，并花更长的时间来交付项目。理解为什么有人需要某个特定的应用程序以及他们将如何使用，往往能得到更好的解决方案。这是一个极端的例子，但它证明了这一点。

5.1.2 理解价值从何而来

理解价值从何而来，除了可以帮助我们设计出更好的解决方案以外，还可以极大地帮助我们排列优先级。在一家美国大型保险公司，rob Park的团队只在较高层次的功能级别上排列优先级，他们不用在较低层次的故事级别上做同样的事，这可以节省很多时间。Park说：

“我们以高层次的角度来描述商业价值和功能的核心。我们把功能分解成尽可能小的故事。举个例子来说：为**14**个州以**PDF**格式提供保险证明。尤其从商业角度来说，我尝试推行‘这有多大的价值，用美元来衡量’的方法。在这个例子中，我们有一个资深成员说：**‘50%**的电话来访是这个目的，其中**50%**的来访是为保险卡的证明而来的，所以只有**25%**的来访是这个目的。’他们知道每天有多少电话来访，他们也了解自动生成**PDF**相比以前人工复制粘贴能节约多少时间，所以事实上他们可以给出具体的数据，这是非常棒的。**”**

→比起只在故事层次上进行讨论，在目标层次上进行讨论能使团队更加高效地处理范围和优先级。

一个可以从中受益的方面就是估算工作量。rob Park的团队发现，讨论目标使他们不必浪费时间在单个故事的估算上：

“我们真的不想在估算故事上花费太多心思。如果你开始估算故事，使用斐波那契数列，你很快就会意识到，对于一个迭代来说要交付任何大于或等于**8**个点的故事，它都太大了，所以我们只用**1**、**2**、**3**和**5**。之后到了一个新的水平，你就会说**5**个点也很大。现在全部都是**1**、**2**和**3**，它们其实都差不多的。我们可以把较高层次的故事分解成大

小都差不多的故事，不用去估算它们，只需要衡量一下需要多少时间，看什么时候可以交付就行了。”

在Software by Numbers一书中，Mark Denne和Jane Cleland-Huang讲述了一个正式的排列优先级的方法，由商业价值驱动，把范围分解成最小可销售的功能。以我的经验来看，预测某个东西能赚多少钱和预测实现这个功能需要多长时间一样困难，并且很可能不准。但如果你的领域使你可以给这些功能标上数值，这将帮助你让商业用户参与进来。让他们来排列功能甚至是商业目标的优先级，比让他们给具体的故事或任务排列优先级要容易得多。

5.1.3 了解商业用户预期的输出是什么

当很难确定目标的时候，预期系统的输出是比较有用的出发点：研究为什么需要这些输出以及软件如何提供这些结果。一旦你明确了期望的输出，就可以专注于实现这些输出背后的需求。分析为什么需要这些输出结果可以帮助你构想出项目的目标。

→我们应该从输出结果的实例开始着手，而不要和商业用户一起讨论如何往系统里增加某些东西。这将帮助商业用户加入到讨论中来，并让他们对系统的输出有一个清晰的认识。

Wes Williams曾经在Sabre参与过一个项目，当时由于用户界面开发的延迟导致了很多返工：

“在客户看到图形界面之前，领域（应用层）的验收测试就已经编写好了。界面推迟了大概**4**个月。客户看到的界面，与他们想象的完全不一样。而当我们开始为**UI**编写测试时，客户所要求的测试数量远远比领域（层）的多。因此领域代码必须做出修改，但客户却认为那部分工作已经完成了。他们认为测试已经有了，而且运行也通过了，所以该部分工作就算完成了。”

对系统预期的输出结果可以帮助我们发现目标，并可以帮我们确定到底需要构建什么内容来提供支持。甚至在未做敏捷项目之前，Adam Geras就已经通过这个思想关注于构建正确的东西：

“我们在很多项目中使用了一种称为‘报表优先’的方法，但它只适用于史诗故事的级别，而且我们主要是在**ERP**实施里使用这种方法，不是敏捷项目。这种方法非常适舍我们，因为找出报表中某项缺失的数据元素要经过大量返工。我们通过优先考虑输出结果来避免此类返工。”

从系统的输出来获取范围的想法出自于BDD社区。这个想法最近受到很多关注，因为它解决了一个普遍存在的问题。在我早期的许多项目中，我们着重关注处理流程，并且一开始就把数据放入到系统中。我们

将处理过程的最终结果（如报表）放在了最后。这种方式的问题是商业用户要到能看到可视化输出结果的时候才参与进来，这样常常导致返工。从输出部分开始工作，能确保商业用户可以一直提供反馈。

5.1.4 让开发人员提供用户故事的“我想要”部分

适用于：商业用户信任开发团队的时候

uSwitch的团队与他们的商业用户一起定义用户故事。商业用户给出故事中的利益相关者和期望的价值，开发团队给出隐含解决方案的部分。在标准的用户故事格式中，这意味着商业用户提供“作为……”和“为了……”语句，开发人员提供“我想要……”语句。

用户故事		
	商业用户	开发人员
作为	X	
我想要		X
目的是	X	

→目标中获取正确范围的一个好方法，就是坚定不移地把提供解决方案的责任交付给开发团队。

如果你有幸可以在高层次的级别上控制项目的范围，请确保开发人员和测试人员都参与到讨论中，并使解决方案着重关注实现已明确定义的商业目标。这将消除今后大量不必要的工作，同时为协作制定需求说明做好准备。

用户故事的组成部分

用户故事描述了用户如何从系统中获得特定的价值。团队一般使用用户故事来做计划并对短期工作的范围进行优先级排列。用户故事通常由以下**3**部分组成：

作为利益相关者；

为了实现某件有价值的事情；

我想要某个系统功能。

举例来说，“作为市场经理，为了能够直接向客户推销产品，我想要系统在客户注册忠诚度计划时记录客户的个人信息。”

不同的作者会把这**3**部分按不同的顺序排列，但是这**3**部分都是需要记录的。本书为了不同的目的会对用户故事的这些部分采用各种不同的顺序。

5.2 在没有高层次控制权的情况下，协作确定范围

对大多与我共事过的团队，尤其是那些大公司的团队来说，项目范围是从上级部门传递给他们的一种东西。许多团队认为，当他们只维护大型系统的一小部分时，讨论商业目标是不可能的事情。然而即使是这

种情况，了解商业用户试图实现的目标也可以帮助你使项目专注于真正重要的事情。

以下是一些提示，它们将有助于在没有项目高层次控制权的时候高效地进行协作确定项目范围。

5.2.1 询问“为什么这些东西有用？”

Stuart Ervine曾为一家大型银行开发过一个后台办公应用程序，它可以让商业用户以树状结构来管理交易对方的关系——这个例子就是很典型的“大型系统的一小部分”。即使是在这种情况下，他们还是成功战胜了被分配任务的命运，并获得了真正的需求。

Ervine团队的任务是提高树状层次的性能，这听起来像是一个真正的业务需求，具有明显的效益。但他们团队无法单独重现任何性能问题，因此任何重大的改善都需要修改底层的东西。

他们向用户询问如何提高性能才会有帮助。结果发现用户要自己遍历层次结构，增加账户余额，以此来手工执行复杂的计算。由于交易对方数量很多，他们必须不断地在用户界面上展开折叠树枝，并增加账户余额——这是一个缓慢而且容易出错的计算过程。

他们没有去提高树状层次的性能，而是为用户自动化了整个计算过程。这让计算过程几乎变成了即时计算，而且显著减少了出错的可能性。这个解决方案带来了更好的结果，并且成本也比原先所需的解决方案更低廉。

→我们应该问询高层次的实例，说明某个功能如何产生实际价值，而不是去问询技术上的功能需求说明。这将指引我们找到真正的问题。

在Bridging the Communication Gap一书中，我建议询问“为什么”，并不断重复这个问题，直到答案开始提及金钱。现在我认为，为了得到相同的结果，一个更好的方法是要求提供实例来解释某个功能如何带来帮助。提出“为什么需要这些东西”这样的问题，听起来有点质疑的口气，这会让他人产生防御心态，特别是在大型组织里更是如此。反之，询问“为什么这些东西有用”，这种方式以讨论的口吻开始，便不会挑战任何人的权威性。

5.2.2 询问替代方案

除了询问某些东西如何带来帮助以便获得真正的商业目标，Christian Hassa的建议是讨论替代方案。Hassa解释道：

“有时候，大家解释某个功能的价值时，仍然会有困难（即便是要求提供实例时）。更进一步，我会让他们举出一个例子，说明如果系统不提供某个功能，他们会怎么做（临时解决方案）。通常这会帮助

他们表达出特定功能的价值。”

→从商业角度发现其他选择，一个好的策略是寻求替代方案。

寻求替代方案可以让功能的提出者再次进行思考，所提议的解决方案是否是最佳方案。同时，也应该与交付团队一起讨论替代方案。

5.2.3 不要只顾最低层次的需求

由于要让交付内容瘦身以便适合一个迭代，许多团队会将产品功能清单分割成更低层次的东西。虽然这样有助于简化工作流程，但也可能让团队忽略了大局。

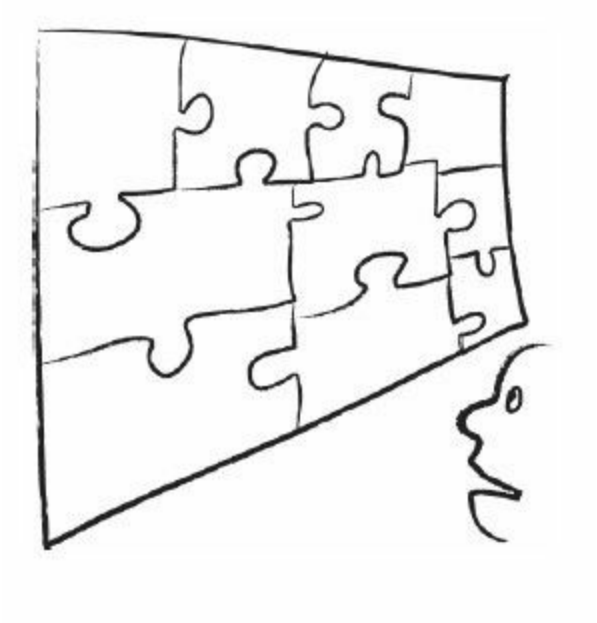
→作为一个过程，实例化需求说明对于高层次的故事和低层次的故事都是有用的。一旦我们获取到高层次的例子，可以说明某个功能如何产生作用，我们就可以将其捕获成高层次的需求说明。这种高层次的例子让我们可以客观地权衡是否交付某个功能。

Ismo Aro参与过诺基亚西门子的一个项目，因为他们没有高层次的需求说明，他的团队受到了挫折。他说：

“用户故事必须适合**sprint**。当一系列的此类故事完成时，就对它们进行独立测试。更大的用户故事实际上没有进行测试。当用户故事的粒度较小的时候，你无法从产品功能清单上判断某项功能是否真的完成了。”

将较大的用户故事分割成较小的可以单独交付的故事是一种很好的实践。为了了解故事何时完成，我们仍然需要查看较高层次的故事。为了兼顾高层次的故事和低层次的故事，我们需要一个分层的功能清单，而不是扁平的、线型的功能清单。

较低层次的需求说明和测试可以告诉我们已经交付部分的逻辑是否正确；而较高层次的验收测试则可以告诉我们那些部件是否按预期方式工作。



5.2.4 确保团队交付完整的功能

适用于：大型多站点项目

Wes Williams把5.1.3节中所描述的问题归咎于团队的分工。不同的团队交付系统的不同组件（这里指的是领域层和用户界面），这让划分工作变得很困难，难以让每个团队与他们的用户一起讨论期望的输出。因此他们重组了团队的工作，让他们可以交付完整的功能。Williams评论道：

“我们用了半年左右的时间组建起了功能团队。这带来了非常不同的结果，尤其是在某种意义上说，它消除了一些相同的工作、大量的重复劳动以及很多返工。幸运的是，我们已经有很多测试可以协助我们这么做。我们有时必须回过头去加入一些新的功能，但那大多是增加——而不是修改。”

→当团队交付完整的功能时，他们可以更加密切地与商业用户一起设计范围并确定需要构建的内容，这只是因为他们可以与用户讨论完整的功能。更多有关功能团队的信息，请参考电子书**Feature Team Primer** 功能团队入门①。

注释：①http://www.featureteams.org

即使没有项目范围的高层次控制权，团队仍然可以通过以下方法影响项目构建的功能：

对需求积极提出质疑；

了解真正的商业目标；

了解谁需要以及为什么需要何种功能。

虽然结果不如一开始就从商业目标中获得正确的范围来得有效，但这种方法可以避免在后来的过程中进行不必要的返工，同时可以确保商业用户得到他们所需的软件。

5.3 更多信息

目前在这个领域有众多创新，而本书涉及的只是我采访的团队所使用的一些方法。

相关的新兴技术也值得一提，可以写成另一本书。要了解如何从目标中获取范围的前沿技术，以及如何勾画它们之间的关系，请参考以下内容：

特性注入：一种通过高层次的实例，迭代式地从目标中获取范围的方法；

效应映射：一种针对项目范围的可视化技术，层次化分析目标、利益相关者以及功能；

用户故事映射：一种提供全局概览的用户故事层次化映射技术。

遗憾的是，关于以上新兴实践的出版物非常少。据我所知，唯一提及特性注入的出版物是一本漫画②，次优的材料是来自Chris Matts的有关Picasa的笔记的一些扫描件③。唯一涉及效应映射的出版物是一本用瑞典语写的书，英文译本为Effect Managing IT④，其翻译质量很差；此外我在网上也发布了一本相关的白皮书⑤。Jeff Patton的博客提供了很多关于被动和主动获取范围的内容⑥，相当不错，同时他正在编写一本关于敏捷产品设计的书，希望能更多地涉及该领域。

注释：②到www.lulu.com/product/file-download/real-options-at-agile-2009/5949486可免费下载。

注释：③http://picasaweb.google.co.uk/chris.matts/FeatureInjection#

注释：④Mijo Balic和Ingrid Ottersten的Effect Managing IT(copenhagen Business Schooll Press，2007)。

注释：⑤http://gojko.net/effect-map

注释：⑥www.agileproductdesign.com

5.4 铭记

当需求作为任务分配给你时，先停一停：请获得必要的信息，以便理解真正的问题；然后进行协作设计解决方案。

如果无法避免任务式需求，请先想办法获取到高层次的实例，解析需求如何起作用。如此将有助于理解谁以及为什么需要这些需求，这样你才能设计解决方案。

为了获取适当的范围，请考虑一下当前这个里程碑的商业目标，想一想哪些项目干系人可以出一份力，又有哪些项目干系人会受到这个里程碑影响。

以系统的输出为出发点，可以让商业用户更多地参与进来。

把组件团队改组为能够交付完整功能的团队。

调查新兴技术，包含特性注入、用户故事映射以及效应映射，以便

有效地从目标中获取范围。

第6章 通过协作制定需求说明

实例化需求说明在概念上与传统的需求说明或测试过程是不一样的，特别是它基于协作的方式。如果我们独自编写文档而不寻求协作，那么即使采用了本书介绍的其余所有模式，实例化需求说明也无法发挥作用。

在Bridging the communication Gap一书中，我专注于将大型的、团队全体参与的需求说明工作坊作为协作式需求说明的主要工具。而在编写这本书的过程中，我最大的一个感触是现实的情况要比前一本书介绍的复杂得多。针对协作式需求说明，不同的团队在不同的环境中都会有他们各自的方式，即使来自同一组织的团队，其协作方式也不尽相同。

在本章中，我将介绍协作式需求说明最常见的模型，包括大型工作坊、小型工作坊以及最流行的工作坊替代方式。这将帮助你理解各种协作式需求说明方法的优缺点。我还会介绍一些很好的实践方法，用于准备协作，同时还有一些很好的想法，可以帮助你为团队选择合适的协作模型但是我们首先要面对的问题是到底有没有必要进行协作。

为了更好地介绍一个协作式需求说明的例子，我们还需要回顾一个与此相关的做法：举例说明。你将在本书7.1节中看到如何举办一个需求说明工作坊的例子。

6.1 为什么需要协作制定需求说明

协作制定需求说明是一个非常好的方法，可以对我们需要完成的内容建立共识，并确保系统的各个方面都被包含在需求说明中。协作还有助于团队制定易于理解的需求说明和容易维护的测试。

Jodie Parker认为，他们在LMAX实施实例化需求说明的一个最大问题是在制定需求说明时没有进行协作。她说：

“大家没有意识到交谈是多么富有价值。开发人员起初认为测试人员对交谈不感兴趣，因为开发人员太过技术性，而测试人员可以了解如何和代码库打交道、他们可以提出建议告诉你有什么地方可能会对其他测试有所影响，或者需要对语言做出什么修改。测试人员也认为他们太忙了。你只有通过实践才能知道协作式需求说明多有价值。”

即使对软件系统所涉及的业务领域有非常完善的理解（我从来没有见过这样的团队），协作制定需求说明依然是值得的。分析人员和测试人员可能知道需要说明和测试什么，但是不一定知道如何组织这些信息

才能容易地进行自动化并驱动开发——程序员却知道。Marta Gonzalez Ferrero曾参与过一个项目，起初测试人员自己编写了所有的验收测试，他们并不认为那些是需求说明。她说开发人员经常不知道如何使用这些测试：

“在最开始，测试人员制作**FitNesse**表格并交给开发人员。这么做导致了一些问题，因为开发人员返回来说这些页面很难理解或者很难自动化。之后，他们开始一起工作。**”**

协作制定需求说明和编写验收测试的工作没有做好，最终必然会导致测试的维护成本过高。这是LisaCrispin从测试设计中得到的一个最重要的教训。她解释说：

“无论何时，当我们必须做出更改的时候，我们都有太多的测试（可执行的需求说明）需要修改。如果有很多测试，就很难重构。我应该和开发人员一起结对，让他们帮助我设计测试。我可以很容易地表述出问题；我能看出有什么问题。测试人员知道一些基本概念，比如**‘**不要重复自己**’(DRY)**，但是他们对工具没有良好的理解。**”**

因为Crispin在编写并自动化可执行的需求说明时，没有同开发人员进行协作，她编写了太多不容易长期维护的需求说明。

我采访的许多团队在早期都犯了类似的错误。当开发人员独自编写需求说明时，这些文档最终和软件设计绑定得太过紧密，不容易理解。如果测试人员独自编写需求说明，这些文档又会组织得不好，难以维护。相比之下，成功的团队碰到此类问题后会很快转换到更注重协作工作模式上去。

6.2 最热门的协作模型

虽然我采访的所有团队都通过协作制定需求说明，但是他们协作的方式却是五花八门，从大型的全体工作坊到小型工作坊，甚至于随意的交谈。下面是一些最常见的协作模型以及团队从中获得的好处。

6.2.1 尝试大型的全体工作坊

适用于：从一开始就使用实例化需求说明时

需求说明工作坊是对领域和范围进行密集的、亲自动手的探索实践，它确保实施团队、商业利益相关者和领域专家对系统功能建立一致的共识。我在Bridging the Communication Gap中对此有详细的描述。工作坊确保开发人员和测试人员有足够的信息来完成他们当前迭代的工作。

→团队全体参加的大型需求说明工作坊是建立共识并获取实例最有效的途径之一，那些实例对功能进行了详细的描述。

在工作坊中，程序员和测试人员可以了解业务领域。商业用户开始理解系统的技术约束。由于团队全体的参与，工作坊可以有效地利用商业利益相关者的时间，并且以后不再需要做知识传递。

uSwitch的团队一开始使用需求说明工作坊来帮助实施实例化需求说明。Jon Neale这样描述其效果：

“它对商业人员梳理不大清晰的思路特别有帮助。例如，如果有人尝试申请低于一定金额的贷款，相比一般的贷款申请而言，这完全是另一个场景。直到最后一分钟仍然有大量的业务规则没有涉及。

需求说明工作坊帮助了他们事先考虑这些场景，也帮助我们做得更快。它还有助于开发团队更容易地与其他人进行交流。这样的前题讨论使得整个过程更加顺利——其立竿见影的效果就是有了更多的沟通。**”**

在**PBR**工作坊中实施需求说明工作坊

产品功能清单精炼工作坊**(Product Backlog Refinement，PBR)**是实施好**Scrum**过程的一个关键元素。同时，我发现大多数声称在运用**Scrum**的团队实际上并没有**PBR**工作坊。**PBR**工作坊一般由团队全体参加，它的主要工作包括对功能清单上优先级最高的项目进行分解，并对其进行详细分析以及重新估算。在**Practices for Scaling Lean and Agile**[①]一书中，**Bas Vodde**和**Craig Larman**建议**PBR**工作坊应该占据每个迭代**5%～10%**的时间。

注释：①Craig Larman和Bas Vodde，Practices for Scaling Lean ＆ Agile Development Large，Multisite，and Offshore Product Development with Large-Scale Scrum(Pearson Education，2010)。

在成熟的**Scrum**团队中开始实施实例化需求说明有个简单的方法，就是在产品功能清单精炼工作坊中举例说明需求。这不需要额外的会议或特别的安排。只是**PBR**工作坊的中间部分有所不同而已。

Pyxis技术公司的**Talia**团队就像这样举办他们的工作坊。**André Brissette**这样解释这个过程：

“通常在产品负责人和**Scrum**大师发现功能清单顶部的故事不够详细的时候，团队会举办工作坊。比如，如果有个故事估算为**20**个故事点，他们在**Sprint**中安排一次维护工作坊。我们认为每周或每两周做一次这样的活动是个好习惯，这样可以确保产品功能清单顶部的故事都是容易直接拿来做的。我们查看故事的时候，产品负责人和开发人员对其可行性交换意见。我们在白板上画一些例子，找出技术难点和易用性的问题，然后开发人员会对范围做一个评估或估量。此时我们

使用计划扑克。如果大家对功能范围和所需要的工作量没有意见，那它就这样定下来了。如果发现很难达成一致的意见，我们会试着分解故事，直到所有内容都十分清晰并且大家都对评估的工作量表示认同。”

大型工作坊的准备工作可能会是个噩梦。如果你没有预定好日期，大家可能安排了别的会议，或没有准备好讨论。而定期会议可以解决这个问题。这个做法对想帮忙却又太忙的高层项目干系人特别有用。（提示：致电他们的秘书来安排工作坊。）

如果商业用户或利益相关者没有足够的时间，试着去适应他们的时间表，或者趁他们还在房间里，在产品演示过程中进行需求说明。当商业用户和交付团队不在一个地方工作时，这种办法也同样管用。

大型工作坊可以有效地进行知识传递，并建立整个团队对需求的共识，所以我强烈建议那些打算使用实例化需求说明的团队采用大型工作坊。另一方面，这会消耗大家大量的时间。一旦有了成熟的过程并且团队建立起了领域知识，你就可以采取其他更简单的方式。

6.2.2 尝试小型工作坊（“神勇三剑客”）

适用于：需要频繁地澄清领域问题时

如果领域逻辑很复杂，程序员和测试人员需要频繁澄清，那么由一个人单独负责编写测试，甚至还包括审核，就不是一个好的方法。

→举办小型的工作坊，由一个开发人员，一个测试人员和一个业务分析师参与。

这样的会议有一个流行的名字叫“神勇三剑客”。Janet Gregory和Lisa Crispin在Agile Testing①一书中提到一个类似的协作模型，叫做“三个人的力量”。（过去我称这样的工作坊为“验收测试三人行”，后来有人抱怨其中有暗讽意味，于是停用。）

注释：①Lisa Crispin和Janet Gregory所著的Agile Testing:A Practical Guide for Testers and Agile Teams(Addison-Wesley Professional，2009)。

“神勇三剑客”会议能够从不同方面获得良好的反馈。与大型需求说明工作坊相比，它不能确保整个团队达成一致的共识，但是它更容易组织，并且不需要预先计划。较小的会议也给参与者的工作方式带去更多的灵活性。组织一个围着一台小型显示器的大型工作坊是没有意义的，应该是三个人可以舒服地坐着，方便地查看一个大屏幕。

要想有效地举办一场“神勇三剑客”会议，与会的三个人必须在领域问题上有相似的理解。如果他们没有达到相似的理解，可以考虑让大家在会前做好准备，而不是随时举办。Ian Cooper这样解释：

“组织三方会议的问题在于，如果团队对领域知识的理解存在不平衡，那么对话将会被领域知识更强的人所左右。这和做结对（结对编程）时碰到的问题类似。领域知识丰富的人倾向于掌控会谈。拥有较少领域知识的人有时会问出一些问题，这些问题可能会有许多有趣的见解。让他们事先做些准备有助于他们在会谈中提出问题。**”**

防止工作坊中遗漏一些信息有个常用的技巧，就是做一个类似最终需求说明格式的记录。像“神勇三剑客”这样较小的组，只要你手头有一台显示器和一个键盘，就可以创建一个这样的文件。Rob Park曾在一家美国大型保险公司的团队中工作，他们使用“神勇三剑客”来进行协作。Park说：

“‘神勇三剑客**’**会议的结果是一份实际的功能文件**——**假定**—**当**—**那么**(Given-When-Then)**。我们不必关心测试装置或下面的其他层，而验收条件就是结果。有时不必非常精确**——**例如，我们知道最好要有一个真实的保单号，我们会先在笔记本或某个地方记录下来，而后再做整理。但是主要的需求是在为功能编写代码之前，要完成我们一致认同的测试，至少内容要完成。**”**

TraderMedia公司Stuart Taylor的团队对每个故事都有非正式的沟通，然后产生相应的测试。由开发人员和测试人员一起完成。Taylor解释这个过程说：

“当要开始一个故事的时候，开发人员会叫来**QA**说：**‘**我准备要做这个故事了。**’**然后他们会商量如何做测试。开发人员会谈论他如何使用**TDD**来进行开发。例如**‘**电话号码字段我会用一个整型**’**，紧接着**QA**会说：**‘**那如果我输入**++**、括号或以**0**开头的字符又将如何呢？**’**

QA开始基于业务验收条件编写（验收）测试，利用测试的思想考

虑边界情况。这些测试会给BA和开发人员看。在演示的时候我们应该能够看到它们被执行。”

协作编写半正式的测试可以保证之后进行自动化时不会扭曲一些信息。同时它还有助于分享如何使用例子来编写良好的需求说明；只有当全组人员围着一个显示器和键盘坐着的时候才可行。不要尝试在全体工作坊中起草半正式文档，因为这样不利于所有人都参与。

→对于已经拥有良好目标领域知识和成熟产品的团队，他们不必举行会议或单独的会谈来讨论故事的验收条件。开发人员和测试人员不需要预先为需求说明提供太多信息，他们可以在具体实现的时候解决较小的功能分歧。这样的团队可以用非正式谈话或评审的方式来协作。

6.2.3 结对编写

适用于：成熟的产品

即使在某些情况下，开发人员有足够多的信息而无需大型工作坊就可以开展工作，团队依然发现协作编写带实例的需求说明非常有帮助。

→分析师可以提供正确的行为，而开发人员知道编写测试的最佳方式，这样之后就可以很容易地进行自动化并融入到活文档系统的其余部分中。

在BNP Paribas，Andrew Jackman的团队正在参与一个比较成熟的产品。他们使用过不同的方式去编写测试，并得出结论说业务分析师和开发人员都需要参与到测试编写中。Andrew说：

“开发人员编写测试的时候，很容易误解故事的内容。如果没有与业务分析师交流，那只是开发人员的单方面见解。而如果让业务分析师编写测试，又完全是另外一回事了。当他们编写故事时，这个故事可能会影响许多已有的测试，但是他们却无法预见到。业务分析师喜欢在一个测试中展现单个故事的流程。一般来说，这会导致大量的重复，因为流程的大部分是一样的。因此我们把部分流程挪到他们自己的测试中。**”**

有些团队，特别是那些没有业务分析师或业务分析师是瓶颈的团队，我们就让测试人员和开发人员一起结对编写测试。这能够让测试人员了解到哪些会被可执行的需求说明覆盖到，同时帮助他们理解哪些需要单独检查。SongKick的团队就是一个很好的例子。Phil Cowans这样解释他们的过程：

“并不是**QA**为开发人员编写（验收）测试；而是他们一起来完成。**QA**在功能交付之前一直负责需求说明，表现为**QA**负责测试计划。开发人员和**QA**一起编写功能文件（需求说明），**QA**会建议应该

覆盖哪些部分。**QA**寻找功能文件的漏洞，指出未被覆盖的功能，同时也会编写手工测试用的测试脚本。”

要想同时考虑测试的几个不同方面并避免井蛙之见，结对编写需求说明是一种廉价而又有效的方式。它能让测试人员了解编写需求说明的最佳方式，以便让需求说明更容易自动化，同时还可以让开发人员了解那些需要特别注意的、有较高风险的功能区域。

6.2.4 让开发人员在迭代开始前频繁地审查测试

适用于：分析师编写测试时

→让资深的开发人员审查需求说明。

Bekk咨询公司和商业用户一起研发挪威的奶牛记录系统，他们编写验收测试的时候没有与开发人员一起工作，但是经常让开发人员来审查他们的测试。据Bekk公司的资深开发人员Mikael Vik说，这种方法给他们带来了同样有效的结果：

“我们总是和他们（商业用户）密切合作来定义**Cucumber**测试。当他们拿着用户故事开始编写**Cucumber**测试时，他们总是会来问我们这些测试是否可行。我们告诉他们如何编写步骤，同时还会给他们一些建议，让他们了解如何扩展**Cucumber**领域语言以便更有效地表达测试的意图。”

如果开发人员没有参与编写需求说明，他们可以花更多的时间来实现功能。注意这会增加风险，导致需求说明缺少实现所需的全部信息，或者难以自动化。

6.2.5 尝试非正式交谈

适用于：商业项目干系人随时都在的时候

如果团队有足够奢侈的环境，商业用户和项目干系人都坐在附近（并且有空回答问题），他们就可以通过临时的非正式对话获得不错的效果。任何与故事相关的人员都可以在开始实现故事之前进行简短的会谈，而不用举办预先安排的大型工作坊。

→只要有任务相关的人员参加非正式的交谈就足够了，这样就可以对所需完成的内容建立起清晰的定义。

“任何相关人员”包括如下几种：

调查故事的分析师；

实现故事的程序员；

做手工探索性测试的测试员；

最终从结果中获利或使用软件的用户，商业项目干系人。

非正式交谈的目的是要确保参与其中的所有人对故事都有相同的理解。在LMAX，这样的交谈出现在一个Sprint的前几天。Jodie Parker解释道：

“根据实际情况决定是否需要交谈。你已经有了自己的想法和草图，并且确实知道如何实现它。如果还未编写验收测试，开发人员和测试人员可以一起结对来写。如果没有进行交谈，那么最后只是‘做了’而非‘做对了’。**”**

有些团队，比如像uSwitch.com的团队，他们没有试图一下子确定所有的验收标准，而是确立了一个通用的底线，并且给测试人员和开发人员足够多的信息，以便他们开始工作。因为和商业用户坐得很近，他们可以在需要时随时展开短暂的交谈（详见第12章）。

有些团队根据故事引入的变更的类型来决定是做非正式的讨论还是进行大型的需求说明工作坊。诺基亚西门子的IsmoAro使用下面这种方法：

“我们同意使用**ATTD**测试用例（需求说明），而无需开会。如果团队觉得这样做容易些，他们可以不用开会。如果故事的实现难度看起来比较大，需要其他项目干系人的意见，他们也可以组织一个**ATDD**会议（需求说明工作坊）。这样做的前提是团队对领域知识很了解。当你要在一个老的功能上加点东西时，就更容易找出测试用例。**”**

6.3 准备协作

协作制定需求说明是一种非常好的方法，可以确保大家达成共识，并可以帮助我们考虑到错综复杂的、独自思考时无法想到的细节。如果讨论的话题需要大量的事先分析，或团队成员的知识水平参差不齐，那么从头开始讨论就是低效并且令人沮丧的。要解决这个问题，很多团队引入了一个准备阶段，如图6-1所示，以此确保功能描述得足够详细，从而有利于进行一场高效的讨论。

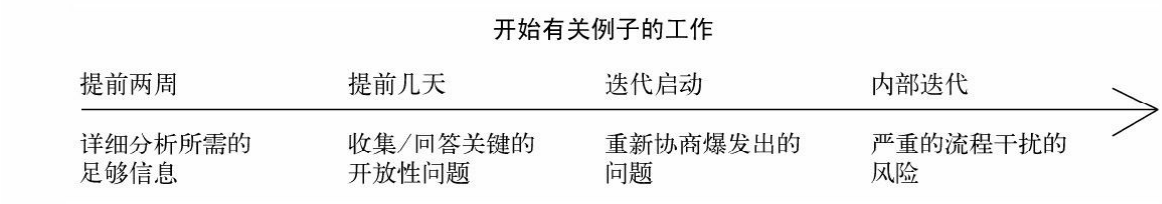


图6-1 通常根据团队何时开始编写例子，可以把它们归纳成4组。那些需要更多时间来分析和讨论开放性问题的团队可以早点开始

准备工作包含与上游项目干系人一起准备一些初始的例子与分析。根据团队成员的具体情况，可以由一个人(通常是分析师的角色)或者一小部分资深人员来完成。

6.3.1 举办介绍会

适用于：项目有众多项目干系人时

有众多项目干系人的团队（例如，当软件使用者涉及公司多个部门，或者软件需求是由多个外部的客户所驱动的），一般来说你需要在迭代开始前几天举办介绍会。有些团队把这个会议叫做预先计划会议。

→介绍会的目的是为即将要做的故事收集一些初始的反馈，并从计划中筛选掉一些需求模糊不清的故事。

介绍会的目的并不是为了得到精炼的需求说明，而是给团队提供足够的时间，对那些可以快速定位的关键问题来收集外部反馈。这不是迭代计划会议或Scrum计划会议。在sprint开始前几天举办介绍会，团队就有机会在真正提炼需求说明或计划会议之前和外部的项目干系人一起讨论一些开放性的问题。

许多团队在介绍会上定义高层次的验收标准，使用简单的列表而非详细的实例。通过找出测试的基础用例，可以帮助我们关注以后的工作。

有些小团队，如ePlan Services的团队，开发人员、项目干系人、项目经理和产品负责人都会参加这个介绍会。对于大型团队或团队群组，只会有小部分人参加。在天空网络服务部门，有6个团队，每个团队会派两三个人参加。

6.3.2 邀请项目干系人

协作制定需求说明的过程之所以有效，是因为它让需求说明进入到商业用户和开发团队成员这个集体的大脑中，确保他们对需求说明有相同的理解。

许多团队邀请他们的业务分析师或产品负责人参与讨论，但是他们却没有邀请客户项目干系人。在这些案例中，团队交付的产品都达到了业务分析师或产品负责人的期望，但是这些期望往往不是最终用户想要的。在我看来，业务分析师是交付团队的一部分，而非客户代表。

→要获得最佳效果，实际的项目干系人都要参与到协作需求说明中。他们是真正能做决定的那些人。

当项目有许多利益相关方时，我们往往只通过一个人来获得所有的需求，一般这个人叫做产品负责人。这种做法对确定范围和优先级很有效，但对需求说明却不是这样。在ePlan Services公司，Lisa Crispin的团队就碰到了这个问题。她说：

“产品负责人需要负责所有的事，但是他又不能同时把所有的事情都做好，因为他在做三四个人的工作。没有人能做所有的事情。有时我们需要一个答案才能完成一个故事，但是他无法提供答案，比如，

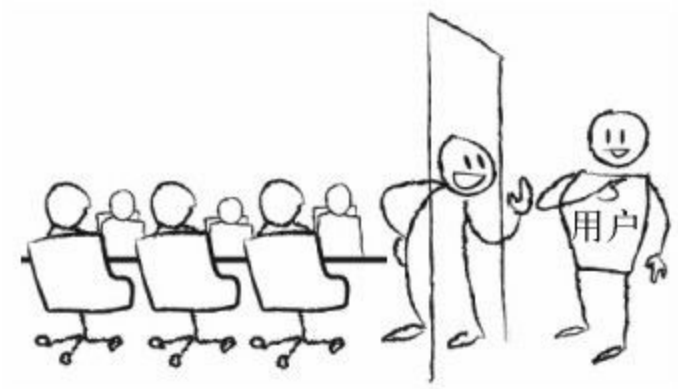
他不理解会计需求。我们还是需要直接和项目干系人交流才能理解这些问题。

他觉得我们绕过了他，这样我们不得不寻找一个平衡点，既要让产品负责人知晓又要从使用该功能的用户那里获取到信息。如果两边存在差异，我们还得让他们一起讨论该问题。”

一个人不可能完全了解所有的事情。但是优先级必须只由一个人来确定，一旦选择了高优先级的故事，团队必须开始和相关的项目干系人一道为特定的故事进行协作，制定需求说明。在Practice for Scaling Lean and Agile一书中，Larman和Vodde区别对待需求澄清和优先级排列。他们认为优先级必须由一个人来决定，但是澄清需求必须由团队自己来完成。

即使团队认为他们十分了解领域知识，自己能够制定出好的需求说明来，邀请最终的项目干系人一起参与还是很重要的。Mike Vogel参与了一个技术数据管理项目，该项目的开发人员对领域知识和技术限制的了解都要比最终用户好。迫于进度压力，他们经常在协作制定需求说明时限制或排除项目干系人，Vogel认为这是他们的一个最大错误。他说：

“起初我们自己做了太多测试创建和验收标准定义的工作。所以我们建立起了元编程，它可以更加快速地推进系统的开发，并让我们处于很大的进度压力之下。但是总会有一些细微之处，我们和客户都不理解，这些都没能在测试中涵盖。”



如果可能，还是要邀请真正的项目干系人进入到协作制定需求说明的过程中。这将保证你能从权威或可靠的来源获得正确的信息，减少预先分析的必要性。

在大型组织中，这可能需要一些说服和政治活动，但这是绝对值得

的。如果你的团队有同样的问题，那么请与产品负责人一起寻找一种可以直接和项目干系人沟通的方式，并且不要去干涉产品负责人作为项目干系人的管理责任。

6.3.3 进行具体的准备工作并事先审查

适用于：项目干系人远离团队时

→项目干系人远离团队的时候，团队至少应该有一个人负责事先准备具体的实例。

在我采访的团队中，先于团队开始工作的人一般是业务分析师或测试人员。他们和项目干系人一起分析需求，商量实例的结构，为最重要的用例捕获价值。团队碰到不明确的需求，需要大量的分析和说明时，也需要有一个人来做事先工作。

在大多数团队中，开发人员也会尽早地审查初始的实例从而提供一些技术上的反馈。这保证团队能在早期发现大多数的功能分歧和问题。项目干系人可以事先回答这些问题，这样团队一起审查故事的时候就不会陷入僵局。

许多团队刚开始时就在这一步失败了，特别是当采用了固定迭代长度的过程时。一个故事涉及的所有内容都应当在一个sprint或一个迭代中完成，这看起来很合乎逻辑。如果领域比较复杂，将需求说明和开发限制在一个迭代内完成，会导致开发人员频繁地陷入困境。

BNP Paribas的Sierra团队尝试把所有事情都限制在一个迭代内，但他们发现这种方式无法使团队高效地工作。后来，他们的业务分析师尝试在团队开工之前就开始工作。Andrew Jackman说：

“我们的项目经理，实际上是产品负责人，他会事先准备所要完成的故事。他和业务分析师会准备好下一轮迭代所要的故事以备在会上讨论，业务分析师会准备好验收测试。我们过去并不是这么做的，而是当开发人员试着编写（验收）测试时，我们会突然询问一些问题，然后发现我们还缺少分析。”

在迭代之前把初始的实例放在一起，可以让团队成员更好地为一起讨论做好准备。在Beazley，Ian Cooper的团队就使用这样的方式。他们的业务分析师和项目干系人都在美国，但是开发团队却在英国。他说：

“由于产品本身以及我们为美国客户服务，时区成了一个大问题，我们很难接触到客户。业务分析师们只是个代理，他们经常碰到问题无法立刻给出答案的情况。开发人员知道许多领域知识，所以分析师和开发人员是主导人员。测试人员并没有真正参与进来。

我们发现让分析师先过一遍需求再来参加会议，事情将会简单很多。测试人员往往会测试所有的可能场景，询问一些边界用例。测试

人员会获得更多的时间去阅读与理解需求，思考可能出现的问题。这让他们可以更好地参与进来。”

如果项目干系人不能参与到需求说明的协作制定中，那么这将大大提高交付团队误解目标的风险。为了降低风险，远离用户的团队要比可以直接接触商业用户的团队做更多的事先分析。如果要这么做，就需要分析师在上游与商业用户和项目干系人一起进行工作，团队其他成员则需要接管一些分析师的下游工作。

如果决定采用迭代前分析，请确保这项工作由某个专门的队员来做，这样可以避免使整个团队受到牵连，同时也解决了故事必须在一个迭代内完成的问题。

6.3.4 让团队成员尽早审查故事

适用于：分析师/领域专家成为瓶颈时

如果分析师或领域专家成了过程的瓶颈，他们将无法带领团队进行迭代前分析。当项目干系人有办法回答问题或者产品已经很成熟的时候，这或许不成问题，开发晚期不会出现功能分歧。

但另一方面，如果团队发现他们没有足够的信息来编写可执行的需求说明，就得有人能较早地提供分析。这个人不必是业务分析师或领域专家，他可以是测试人员或开发人员。

→开发人员和测试人员可以帮助领域专家减轻负担（当领域专家成为瓶颈时），他们可以做第一遍审查来找出常见的问题。这样将提高团队整体的输出，同时也有助于构建跨功能团队。

Clare McLennan参与了一个网页广告项目，这个项目的项目干系人

在德国，而团队在新西兰——几乎整整12个小时的时差。测试人员扮演了本地分析师的角色。他们不能替客户做决定，所以他们先于团队开始工作。McLennan说：

“为了避免时区问题，我们不得不确保每个故事都要在控制当中。如果测试人员看过后觉得没有问题，那么他们会再找一个程序员去看看，确保程序员也觉得没问题。”

在Ultimate软件公司的Global Talent Management团队，产品负责人很忙，团队其他人员会帮忙做分析工作。2个开发人员和1名测试人员组成的小组会事先审查每个故事，找出需要咨询的问题，为与产品负责人开会做好准备。Maykel Suarez说这种方式帮助他们更有效地利用了大家的时间：

“大一点的团队，比如**17**人左右，他们做决定有很大的压力。解决办法是分小组。现在一个小组（包括**1**个测试人员、**2**个开发人员）能够更快地做出决定。在这些准备会议中，我们会讨论小于两个星期迭代的工作，一般只是两三个故事。因此，每隔**3**～**5**天有**3**个人开一个**15**～**30**分钟的会议并不浪费时间或资源。”

6.3.5 只准备初始的实例

适用于：项目干系人都有空的时候

项目干系人都有空回答问题时，团队不需要花太多的时间事先准备详细的实例。但他们发现，为了在讨论前获取到基本的结构，确定一些初始的实例仍然是非常有用的。

→确定初始的实例能帮助我们获得需求说明的基本结构，并可以让讨论更加高效。

在**Pyxis**技术公司的**Talia**项目中，**André Brissette**经常使用外部客户提供的实例来进行需求说明。他是开发团队的商业项目干系人，也和外部客户一起工作。当客户提出新的功能时，他们把系统应如何工作的实例发给他，这些实例会成为将来需求说明的一部分。

uSwitch的团队和他们的项目干系人在同一地点工作，因而他们无需大量的事先准备。团队里任何人都可以在站立会议中提出一个新的故事，提出建议的人往往会先准备基本的实例。

一开始就准备好初始的实例可以使讨论更加高效，因为当团队说明一个需求或确定关键属性时，不必尝试使用结构最好的实例。他们可以更加专注在如何理解并扩展初始的实例上。

6.3.6 不要让过度的准备阻碍了讨论

准备阶段应该是为了让协作更加高效，而不是用来代替协作的。因为测试人员是从组合功能性回归检测的角度来看待可执行的需求说明，

所以有些团队事先准备的信息太多了，他们找出了测试中输入参数所有可能的组合。

→复杂的需求说明难以理解，所以大多数人在这些需求说明中发现不了功能分歧以及不一致的地方。

使用复杂的需求说明，事先分析的效果就如同传统需求从分析师传递到开发人员一样。开发人员只是拿到需求，而非协作构建共识，这将导致误解的产生，而且很有可能到过程晚期才发现功能分歧。

在LMAX，Jodie Parker的团队准备过头了，最终得到的实例看起来很完整。这使得他们直接跳过了讨论，最终导致了需求说明的功能分歧。Parker建议事先准备的实例只要做到“刚刚好”就行了：

“因为我们都是第一次接触这个过程，一开始我们的开发人员说没有足够的信息进行工作。然后业务分析师非常完整地说明了一切，我们的手脚就被束缚了。当开始照着卡片进行开发时，完全没有创新，无法使用更简单的解决方案，因为需求说明规定得太多了。

如果你看了一张卡片后说‘好吧，我完全理解了’，然后你就胸有成竹地去工作，你可能已经做了一百万个假设。如果你看了一张卡片，心里全都是‘我不是很确定’，这会促使你去沟通，在迭代开始时提出来，然后讨论各种不同的实现及其效果。然后，测试人员会考虑它会如何影响测试；业务分析师会考虑后面的问题，看看是否行的通。‘刚刚好’意味着开发人员、业务分析师和测试人员都站在白板前，讨论应该如何实现。”

不管你是否决定让某个人提前花一个星期去准备初始的实例，或准备一个介绍会议来找出问题，请记住，它的目的是为了之后的讨论做准备，而不是为了替代它。

6.4 选择协作模型

我并不认为有一种放之四海而皆准的原则能帮助你为团队选择最好的模型，包括实现个人的事先工作与更多的动手协作之间的平衡。在比较了使用类似过程的团队后，我认为你的决定可以基于以下条件：

产品的成熟度如何？

团队拥有多少领域知识？

典型的更改需要多少分析？

商业用户和开发团队有多近？他们是否有空讨论并验证实例？

过程中的瓶颈在哪儿？

不成熟的产品需要大型的工作坊和大量的事先分析。对于不成熟的产品，让测试人员和开发人员更积极地帮助定义需求说明很重要，因为

底层系统变化频繁，而这些人员拥有商业用户所没有的洞察力。

成熟产品可以有较少的事先分析，可以考虑其他的协作模型。成熟产品可能意味着较少的意外。业务分析师和产品负责人大多十分了解技术能给他们带来什么，而且他们可以事先很好地准备实例。

如果团队比较新或测试人员和开发人员对商业领域的知识了解得不够，那么做大型工作坊就比较值得。在把商业领域知识有效传授给整个团队方面，全体工作坊是一个非常好的方式。一旦团队更好地理解了商业领域，那么小型的、比较专注的讨论可能就足够了。

如果典型的更改需要大量的分析，那么应该有个担任分析师角色的人先于团队与项目干系人一起准备详细的实例。否则，工作坊中的讨论都会结束得很快，并留下大量未解决的问题。如果要开发的是相对较小且能够被充分理解的功能，那么事先准备一些基础实例来让讨论更加流畅，可能也就足够了。

相比有商业用户随时有空回答问题的团队，那些没有与商业用户在一起的团队通常需要更多的事先工作。如果商业用户根本无法参加需求说明工作坊，那就需要事先找出更多的问题与功能分歧，并加以处理。

最后，为已经成为过程瓶颈的团队成员增加更多的工作是完全没有意义的。测试是瓶颈的团队应该让开发人员和业务分析师更多地参与到事先工作中。类似地，业务分析师或领域问题专家是瓶颈的团队应该让测试人员帮助完成事先的分析工作。

6.5 铭记

实例化需求说明非常依赖于商业用户和交付团队成员之间的协作。

交付团队的所有人都要对需求说明的正确性负责。开发人员和测试人员必须提供关于技术实现和验证方面的建议。

大多数团队协作制定需求说明分为两个阶段：有人事先准备功能的初始实例；然后该功能的项目干系人进行讨论，并添加实例来澄清或完成需求说明。

事先准备工作与协同工作之间的平衡，要根据以下几个因素来定：产品的成熟度、交付团队的领域知识水平、典型更改需求的复杂度、过程瓶颈以及商业用户是否有空。

第7章 举例说明

例子既能避免歧义又是进行准确沟通的好方法。在日常的交谈和写作中，我们会不假思索地使用例子。当我在Google里搜索“例如”这个词

组时，返回的搜索结果超过了2.1亿页。

使用传统的需求说明时，例子会在软件开发过程中时隐时现。业务分析师通常会从商业用户那里获取到订单、发货单以及报表的样本，然后他们会将其转换成抽象的需求。开发人员会想出一些例子来解释边界情况并向商业用户或分析师作出澄清，而后他们会将其转化成代码，但他们并不会记录下这些例子。测试人员会设计出测试用例，它们实际上就是系统应当如何工作的实例；这些例子仅供他们自己使用，而不会拿来与开发人员或分析师进行沟通。

每个人都会创造自己的例子，且不说这些例子是否完整，连例子的一致性都无法确保。这就是为什么在软件开发中最终结果往往会背离最初设想的原因。为了避免这种情况，我们必须防止不同角色之间出现的误解，只维护一处事实来源。

例子是避免沟通问题的好工具。如果我们自始至终都能够捕获所有的例子，并在分析、开发和测试中一致地使用它们，那么我们就可以避免进入传话游戏中（而导致信息失真）。

Beazley公司引入实例化需求说明的时候，Marta Gonzalez Ferrero是当时的测试负责人。据她所述，开发团队承诺的工作量超出了他们的能力，同时他们发现，在实现开始的时候所获得的信息往往远远不够。他们的迭代周期是6周，而且开发团队和业务分析师身处于不同的大洲，这让事情进一步复杂化。开发人员从业务分析师那里得到的验收条件也比较抽象（例如，“对于这个业务单元，要确保所有正确的产品都能够显示出来”）。在迭代半途中发现缺少某些重要的东西，会严重扰乱产出。某轮迭代结束时，客户说团队交付的东西完全不是他们所期望的。每个迭代的最后一周保留给Model Office①：实际上是一个迭代的演示会议。有一次，Ferrero到美国做Model Office，并且与业务分析师一起工作了两天，对需求进行举例说明。结果，团队在下一个迭代承诺交付的工作减少了20%，最终他们成功交付了自己承诺的部分。

注释：①Model Office类似于在开发阶段所做的小型UAT/验收测试，并会召开演示会议，以便演示已经完成的功能并获得改进意见。——译者注

“团队的情绪也好了很多，”Ferrero说道，“在此之前，（开发人员）在实现的时候会感觉自己是在瞎做，必须等待业务分析师的反馈。”据Ferrero所述，在他们开始对需求做举例说明后，返工量急剧下降。

Ferrero他们并不是唯一有此体会的团队。几乎所有本书有所提及的

团队都证实了这样一个事实：对需求进行举例说明的方法比抽象的陈述说明来得更有效。因为例子是具体的、明确的，所以它们是制定精确需求的理想工具，这就是为什么我们在日常交流中用其来澄清意思的原因。

Gerald Weinberg和Donald Gause在《探索需求》一书中写道，检验需求是否完整的最好方法是试着对其设计黑盒测试。如果我们没有足够多的信息去设计出好的测试用例，那么我们也绝对没有足够多的信息来构建系统。举例说明需求是一种使用足够具体的、可以检验的信息来定义系统应该如何工作的方法。那些用来说明需求的例子就是很好的黑盒测试。

根据我的经验，举例说明需求所用的时间要比去实现它们少得多。比起先试着去实现软件，然后发现信息不够用，举例说明需求时发现信息不够用所花费的时间要少得多。不要去开发一个不完整的故事，这只会让你在迭代中才发现问题，我们在协作制定需求说明的过程中就可以抓出此类问题，此时我们还能够解决这些问题——并且这个时候商业用户还在。

2009年5月，在Progressive.NET教程中，我举行了一个关于实例化需求说明的3小时的工作坊①。大约有50个人参加了这个工作坊，其中大部分是软件开发人员和测试人员。我们模拟了一种常见情况：一位客户要求我们的团队到他们竞争对手的网站去拷贝一些功能。

注释：①详见http://gojko.net/2009/05/12/examples-make-it-easy-to-spot-inconsistencies

我从一个热门网站拷贝了“21点”游戏的规则，并要求参与者使用例子来描述这些规则。虽然需求采集自一个真实的网站而且只有区区一页，但这些需求模棱两可、累赘冗余并且残缺不全。据我的经验，使用WOrd文档来记录需求往往会引发这种情况。

参与者分成7个团队，每个团队只有一个人懂“21点”。工作坊结束后，所有参与者都一致认为讨论实际的例子可以帮助我们找出不一致的地方和功能分歧。通过反馈调查（见下文附注栏），我测算出了共识度。虽然团队中多数人之前从未涉足过该领域，但是7个团队中有6个团队对困难的边缘情况给出了相同的答案。举例说明需求是交流领域知识并确保达成共识的非常有效的方式。在编写本书所采访的众多团队中，我在他们实际的软件项目上看到了同样的效果。

反馈调查

反馈调查是检验一组人员是否对需求说明达成共识的好方法。在

讨论过一个故事之后，如果有人对该故事提出了一个特殊用例，工作坊的主持人必须请参与者写下他们认为系统应该怎样工作。然后比较全组的回答。如果所有人的回答都一致，那么每个人对需求说明的理解就没有出入。如果回答不一致，那么一种有效的作法是按回答的结果分成多个小组，每个小组选出一个人来解释他们的回答。通过讨论可以揭示误解的根源。

举例说明需求的想法很简单，但是实施起来却没那么容易。找到一组正确的例子来说明一个需求是一个很大的挑战。

本章我会先举一个例子，让大家对使用例子的过程有一个了解。然后我会对如何确定一组能正确说明业务功能的例子提供一些好的意见。最后，我将谈及一些举例说明交叉功能的方法以及一些不容易捕获精确价值的概念。

7.1 举例说明：一个例子

为了阐明如何对一个需求进行举例说明，让我们来看看一个关于虚构公司(ACME网店)的例子。这是本书中唯一一个虚构的公司，为了让例子简单，我必须虚构一个。ACME是一家小型网店，它的开发团队刚刚开始了一个需求说明的工作坊。Barbara是一名业务分析师，此前一周，她与公司老板Owen一起花了一些时间收集了一些初始的例子。她主持本次工作坊，并引入了第一个故事。

Barbara：列表中的下一项是免费送货服务。我们已经安排曼宁**(Manning)**出版社对他们的书籍提供免费送货服务。最基本的例子是：如果用户购买了一本曼宁的书，比如说就是你手中这本，购物车将提供免费送货服务。有问题吗？

David是开发人员，他发现了一个潜在的功能分歧。他问道：是免费送到任何地方吗？那如果客户住在南美洲的一个小岛上呢？对其免费送货的费用将远远高于售书所获得的利润。

Barbara：不，不是针对全世界，只针对美国国内。

Tessa是测试人员，她问了另一个问题：在测试该功能时，首先我会检查我们不对所有图书提供免费送货服务。我们可以增加一个实例来说明免费送货服务只针对曼宁的图书吗？

Barbara：当然可以。例如，**Agile Testing**是由**Addison-Wesley**出版社出版的。如果用户买了这本书，那么购物车将不提供免费送货服务。我觉得这比较简单，没有更多需要再提的了。谁可以再另外想一个别的例子？我们能否找出会让这个例子失效的数据？

David：没有数值相关的边界条件，但是我们可以围绕购物车清单

展开。例如，如果我同时购买了**Agile Testing**和你手中这本书将会怎样？

Barbara：两本书都将免费送货。只要购物车中有一本曼宁的图书，你就会得到免费送货服务。

David：明白了。但是如果我同时买了你手中这本书和一个冰箱，那又将如何呢？送货服务的成本将比售书的利润高多了。

Barbara：这会是个问题。我没跟**Owen**讨论到这个问题。我必须确认之后才能回答你。还有其他问题吗？

David：除了这个，没有别的了。

Barbara：好。那除了冰箱这个问题，我们已经有足够多的信息可以开工了吗？

David和**Tessa**：是的。

Barbara：很好。下周初我将回答你的冰箱问题。

7.2 例子必须精确到位

好的例子可以帮助我们避免模棱两可。要做到这一点，必须完全没有误解。每个例子都必须2清晰地定义好上下文以及系统如何在一个给定的情况下工作，并且理想情况下这种描述必须很容易进行校验。

7.2.1 不要在例子中出现“是/否”的回答

适用于：没有单独定义基本概念时

在描述过程时，我采访的很多团队使用了“是/否”的答案，这使得例子太过简单化。这会产生误导，并给大家一个错觉“我们已经达成共识”，而事实上并不是这样。

例如，TechTalk公司在一个基于Web的退货系统中，描述Email提醒功能的需求时碰到了这个问题。他们有一些例子是关于何时发送Email的，但是他们并没有讨论Email的内容。该系统的开发人员Gaspar Nagy说：“客户期望我们包含失败的案例和解决方法，而我们并没有捕获到客户的期望。”

我为一家大型投资银行举行过一个需求说明工作坊。他们的团队在讨论款项支付如何导向到8不同的系统。他们把例子列在表格里，左边是条件，右边是不同的子系统，根据目标是否接受事务来决定将列标成“是”还是“否”。我让他们写下发送到每个系统的消息的关键特性，而不是“是”或“否”。那时，大多数开发人员都误解了的几个有意思的案例出现了。例如，其中一个系统的事务更新需要两条消息，而不是一条：一条是取消原有事务，另一条是注册一个新的事务。

→提防例子中出现“是/否”的回答，试着将其重写得更加精确到

位。

只要潜在的概念已经单独描述过了，还是可以在例子中保留“是/否”。例如，一组例子可以告诉你Email是否已经发出，而另一组例子用来描述Email的内容。

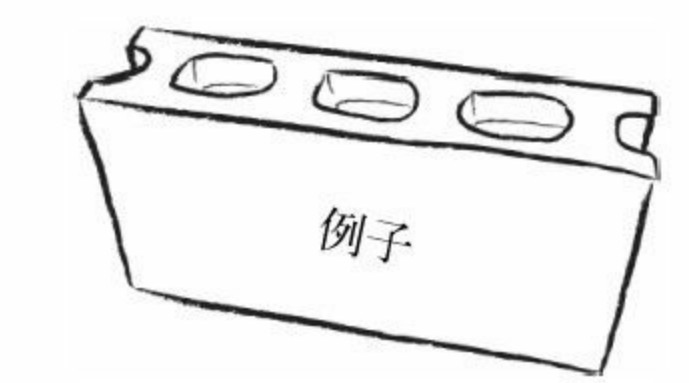
7.2.2 避免使用等价抽象类

适用于：当你可以指定一个具体的例子时

等价类（比如“小于10”）或者变量会让人产生一种达成共识的错觉。不使用具体的例子，不同的人就可能会有不同的理解，例如小于10是否包含负数。

当输入参数使用等价类时，期望的输出必定被定义为将输入值作为变量的公式。这种方式实际上是对功能描述的复制。它没有提供具体的例子来验证，也就失去了举例说明的价值。

为了自动化，这些值的等价类必须转译成具体的某种东西，这意味着无论谁自动化这些验证，都必须将需求说明转译成自动化代码。这就意味着出现误解和曲解的概率增大。



据我的经验，需求里那些看起来很直观的东西最有可能欺骗我们。那些容易混淆的概念会被讨论和探讨，但是对于那些似乎很明确的概念（不同的人往往会对它们有不一样的理解），往往不易察觉，并会造成问题。

→不要使用等价类，要使用有代表性的具体的例子。具体的例子让我们可以无需修改就直接自动化需求说明的验证，并能确保所有团队成员达成共识。

你可以安全地将等价类用作预期的输出，特别是当你描述的过程具有不确定性的时候。例如，指明一个操作的结果必须介于0.1与0.2之间，这样的陈述仍然使需求说明具有可测试性。如果过程具有确定性，那么一个具体的数值可以使其更加精确；即使对于输出，也可以尝试使

用具体的数值。

7.3 例子必须完整

我们必须有足够的例子来描述某个功能的完整范围。主要业务用例的预期行为和简单的例子是很好的着手点，但一般它们并不是我们需要实现的功能的总和。以下的一些想法有关于如何对一组初始的例子进行扩展，以便对功能提供一个全面的了解。

7.3.1 用数据作试验

→一旦你拥有了一组你认为很完整的例子，那么就应该查看例子的结构并试着想出一些会违反规则的输入组舍。这有助于你发现可能漏掉的东西，使得需求说明更健壮、更完整。

如果例子包含数值，你可以试着同时使用不同边界条件附近的较大数字和较小数字。尝试使用零或负数。如果例子包含了多个实体，就考虑一下是否可以使用多个对象、一个不包含实体的例子是否仍然有效，以及如果同一个实体定义了两次又会怎么样。

当协作制定需求说明时，我尤其期望测试人员帮忙寻找此类例子。他们应该会有技术和进一步的试探方法来找出潜在的有问题的用例。

你找出的很多技术上的边缘情况并不会代表有效的例子，这并没有什么问题。除非你正在对无效参数说明错误消息（这种情况下，这些是针对该业务功能的有效例子），否则不要去涉及具体细节。考虑这些不同的情况，你可能会找出一些不一致的地方，以及之前可能还没有考虑过的边缘情况。

用数据作试验的一个风险是输出将会有太多没有显著区别的例子。这就是为什么下一个步骤“提炼需求说明”（详见第8章）如此重要的原因。

7.3.2 使用替代方法来检验功能

适用于：复杂的/遗留的基础设施

在复杂的IT系统里，很容易忘记所有应该发送信息的地方。

→为了测试某个故事是否有一组定义良好的例子，你可以要求商业用户考虑一个替代方案来验证最终的实现。

“你还会怎么对它进行测试？”是开始这类讨论的一个好问题。Bas Vodde还建议像下面这么问：“还会有其他事情发生吗？”在前文7.2.1节里提到的那个需求说明工作坊中，我提出了这个问题之后，我们发现了一个遗留的数据仓库，有些人认为它应该接收事务，而其他人则认为它应该忽视事务。这一发现促使我们进行了一场讨论并结束了这个功能分

歧。

Pascal Mestdach在IHC公司的Central Patient Administration项目中有类似的经历。他们经常碰到问题，客户希望在系统迁移期间，那些新系统里存储的数据会同时发送到遗留系统，但是团队并不了解这些需求。如果当时要求客户提供一种替代方法来测试功能，那他们就可以发现客户期望在遗留系统里同样能看到这些信息。

要求用户使用替代方法来检测功能，也有助于团队讨论出自动化校验的最佳地点。

7.4 例子必须要真实

当我们使用例子来说明功能的时候，模棱两可和前后矛盾的地方就会突显出来，因为例子着重讨论真实的案例而非抽象的规则。为了更好地发挥例子的功效，例子必须要真实。任何编造的、简化的或者抽象的例子都不会有足够多的细节，也无法展示足够多的变化。要特别小心抽象实体，比如“客户A”。找一个真实的客户，他必须具备你要描述的特点，或者你可以关注到特点本身而非客户身上，这样会更好。

7.4.1 避免虚构自己的数据

适用于：数据驱动的项目

→在数据驱动的项目中使用真实的数据十分重要，因为许多事情

可能会依赖于细微的变化和不一致的地方。

来自于Knowledgent Group的Mike Vogel参与过一个新项目，他们使用元数据驱动ETL来填充药物研究的数据仓库。他们使用了实例化需求说明，但是客户和团队双方都使用虚构的例子来说明功能，而没有去寻找真实的数据样本。他说这种方式并没能帮助他们避免不一致的问题。

“他们（客户代表）虚构例子，而没有处理实际的变化。他们假定自己可以做某些事情，并把这些事情排除在例子之外。当实际系统的数据进来时，总是会碰到许多意外情况。”

当项目涉及遗留系统时，这类问题会更严重，因为遗留数据往往违背预期的一致性规则（以及一般的逻辑规则）。

Jonas Bandi曾经在TechTalk公司为学校的数据管理重写了一个遗留系统，在理解了原有的遗留数据结构和关系后，他发现里面十分复杂。他们期望实例化需求说明可以保护他们免受回退(详见4.5.2节)，防止缺陷，但事与愿违。他们基于自己对领域的理解虚构了例子。真实的遗留数据经常有令人意想不到的意外情况。Bandi说：

“即使在所有场景（测试结果）都是绿色的（测试通过）并且一切都看起来很好的情况下，我们仍然还有很多缺陷，这是由来自于遗留系统的数据所产生的。”

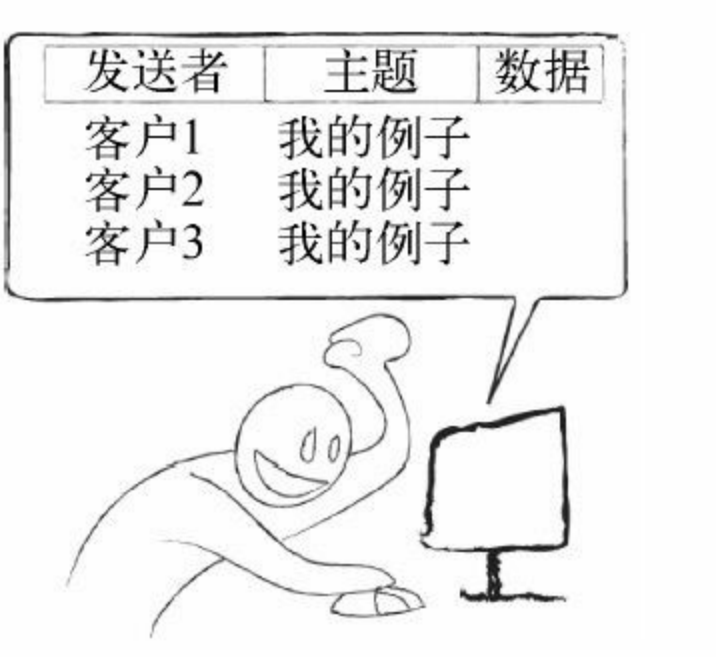
为了减少遗留数据在迭代后期给团队带来的意外风险，请尝试在例子中使用来自于现有遗留系统的真实数据，而非使用全新的用例。

使用现有的数据可能需要对一些敏感数据自动做模糊处理，这会对自动化数据的管理策略带来影响。对于此类问题有一些好的解决办法，详见9.5节。

7.4.2 直接从客户那里获得基本的例子

适用于：与企业用户一起工作时

对于卖企业软件给多个客户的团队而言，他们很少会有客户代表参与到协作制定需求说明的工作坊中，因为这是一种奢求。产品经理会从不同的客户那里收集需求，并决定发布计划。这会导致歧义和误解的可能性。我们可能会有非常精确和清楚的例子，但它们可能并没有捕获客户所需要的。



→要确保用来说明需求的例子是真实的。真实的例子包含来自客户的数据。

当我们与外部的项目干系人一起工作时，我们同样可以应用那些团队内部用来确保共识的窍门。André Brissette把客户的电子邮件当作入手点，用它来展开Talia系统自动对话的讨论：

“他们应该写一封这样的电子邮件：‘如果这个问题我可以问**Talia**，那将会更简单，她会告诉我**……**，然后我就能够去做**……**。’在这个案例中，用户提供了对话的第一个草案。**”**

Brissette记录下此类电子邮件，并把它们作为初始的例子来描述所需的功能。这可以确保外部项目干系人的请求得到满足。请看图7-1，它是由此产生的需求说明的例子。请注意，理想情况下这个例子需要进一步提炼。详见8.3.2节。

在RainStor，Adam Knight的团队使用这种方法开发了一个结构化数据的归档系统。他们与客户一起获取真实的数据集和典型查询所预期的目标。当客户不能提供一个具体的用例时，他们会追问例子，有时候会安排与客户一起进行工作坊。有个常见的例子是关于客户不能提供具体用例的：一个还没有买家的经销商想要系统支持某些功能，因为他们猜想这样会更容易销售。其中一个例子是他们要求电子邮件归档系统提供镜像功能。Knight说：

“他们看到了一个电子邮件归档系统，然后说我们的系统要能够以同样的方式进行工作。电子邮件归档系统有上千封邮件，但是在我们的系统里会有几十亿的记录。你想要同级别的粒度吗？日志要怎么

办？这是一种最难处理的需求。一般来讲，我们会设法追问客户要一些例子。我们会安排一些演示来做功能原型并从头检查一遍。”

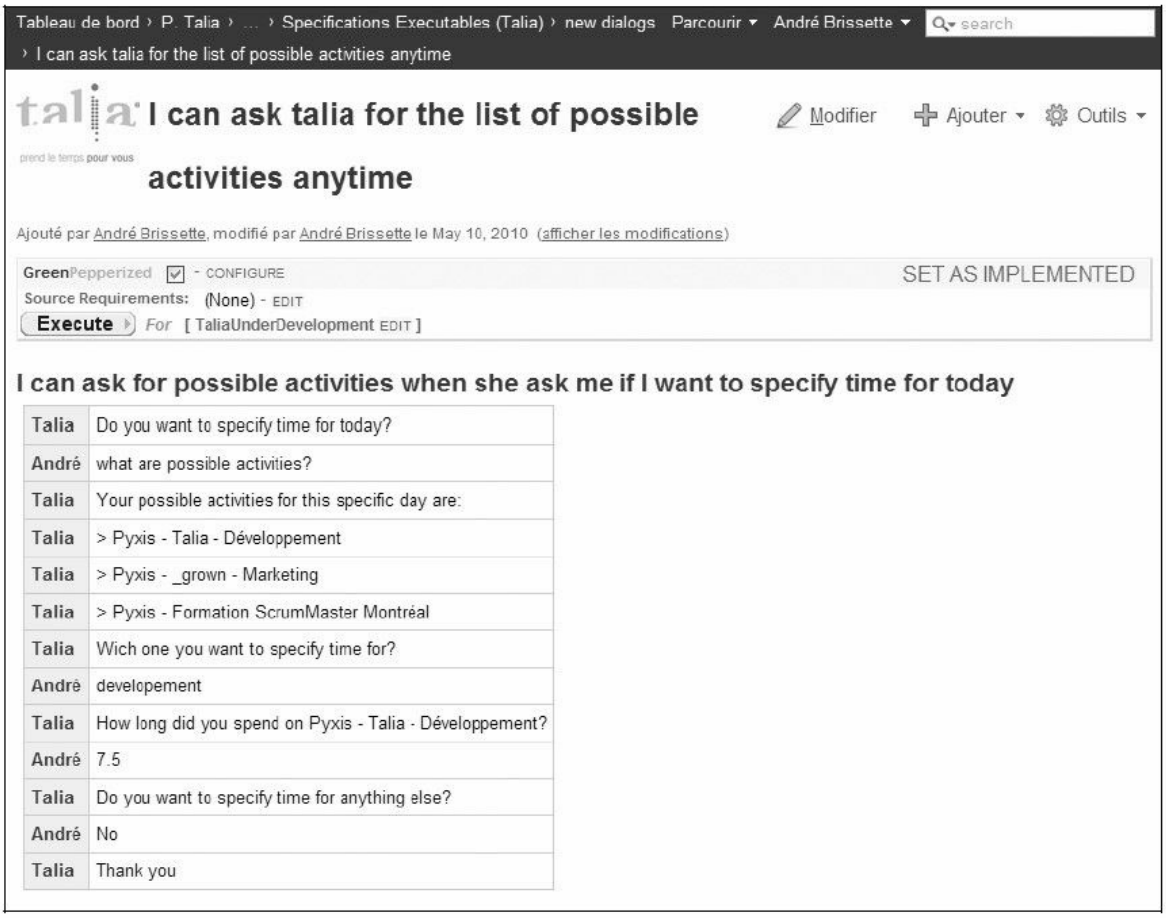


图7-1 一个客户对话的例子，用作Talia系统的一个需求说明

为了避免产品经理对客户需求的理解和客户真正的需求之间存在模糊和误解，请坚持使用例子来与客户沟通。可以在需求说明工作坊上利用这些例子展开讨论。这些例子还必须包含在最终的可执行需求说明里，以便确保客户的期望得到满足。

7.5 例子应该易于理解

刚开始使用实例化需求说明的时候，团队常犯的一个错误是使用错综复杂的例子来描述需求。他们专注于捕获真实例子的细枝末节，并创建庞大的、另人费解的表格，有几十多列和行。这样的例子会增加评估需求说明一致性和完整性的难度。

我更喜欢把例子当成需求，不喜欢抽象的描述，一个主要的原因是

它们可以让我考虑到功能分歧和不一致性的问题。事情一旦精确，就更容易发现缺失的情况。这要求对一个特定功能的整组例子都有所了解。如果例子不易理解，我们将无法评估它们的完整性和一致性。下面是一些既能避免此类问题，同时依然能够保持例子精确性和真实性的想法。

7.5.1 避免探讨所有可能的组合

当团队开始使用实例描述需求时，测试人员通常会误解该过程的目的，并且坚持覆盖参数的所有可能组合。通过实例说明现有的所有情况并没有多大意义，也不会增进理解。

→当举例说明需求时，寻找那些可以推进讨论并增进理解的例子。

我强力反对不经讨论就丢弃任何作为边界情况提出的例子。如果有人提出了一个边界条件的例子，而别人认为这种情况已经覆盖到了，那么可能有两种原因：要么提出这种例子的人不理解现存的例子，要么他们真的发现了一些别人没有发现的、会破坏现有描述的东西。这两种情况都值得我们对这个例子进行讨论，以确保房间里的每个人都对这个作为边界情况提出的例子有相同的理解。

7.5.2 寻找隐含的概念

如果用来描述某个功能的例子太多了，或者所用的例子很复杂，往往意味着这些例子应该用更高层次的抽象来描述。

→应该仔细查看这些例子，并且试着找出缺失和隐含的概念。将这些概念显式化，并单独对它们进行定义。重整这类例子会使需求说明更易理解，并带来更好的软件设计。

领域驱动设计[①]的一个核心思想是找出缺失和隐含的概念，并在系统设计中将其显式化。

注释：①出自Eric Evans所著的Domain-Driven Design:Tackling Complexity in the Heart of Software一书(Boston，AddiSon-Wesley Professional，2003)，中译本《领域驱动设计：软件核心复杂性应对之道》。

我曾经协助过一个团队举行了一个工作坊，当时他们正在重写一个财务子系统，并逐步将交易从遗留系统迁移到新系统上。这个工作坊专注的需求是要将荷兰的交易迁移到新系统上。起初我们在一块白板上写下例子，但很快就找不到空白的地方了。仔细审视这些例子后，我们发现阐述的事情无非有3件：如何判定哪些交易是荷兰的，如何判定哪些交易已经迁移了，以及交易迁移之后会有什么变化。

因为我们同时阐述这几件事情，所以我们需要处理相关情况的众多组合。尝试总结这些例子的时候，我们发现了两个隐含的概念：交易地和迁移状态。之后我们将这个需求分成3个部分，并且每一部分都使用一组单独的、集中的例子进行描述。我们有一个需求说明是关于如何判断一个交易是否是在荷兰进行的（如何计算交易地点）。另外有一组专门的例子描述了交易地点对迁移状态产生的影响。在这组例子中，我们只用了一次“荷兰”，而无须遍历所有构成荷兰交易的情况。第三组例子描述了在处理迁移和未迁移交易时的区别。

这样分割需求说明可以帮助团队显著改善系统的设计——因为3组不同的例子清晰地指出了模块化的概念。下次他们再有迁移一组交易的需求时，只要修改迁移交易的定义就可以了。交易在迁移之后保持不变的话又会怎样呢？同样地，判定交易地点的方法没有变化。

分离这些概念还有助于我们对交易地点展开更加有意义的讨论，因为我们应对的是一组较小而且集中的例子。我们发现，有些人认为股票发生交易的公司的注册地会决定交易地点，而其他人则认为这只与公司在哪里进行股票交易有关。

寻找缺失的概念和提高抽象的层次与日常交流中所发生的事情并没有区别。比如试着不用“车”这个字，给出一句诸如“如果开车来，请事先订好停车位”的指示语。你要注重它的属性。汽车可以描述成是一辆带有4个轮子、4扇门、4个座位以及一个柴油发动机的交通工具。但是我们还有双开门的车，有其他类型的发动机和不同数量的座位等。列出

所有这些例子会使得说明变得极端复杂；反之，我们会创建一个更高层次的概念来改善沟通。汽车是如何制造的与停车指示没有任何关系，重要的是到达的人是否开车来。

无论何时看到一个需求说明里有太多或太复杂的例子，你可以试着提高这些描述的抽象层次，然后另行说明基本概念。

通过使用精确真实的例子来描述需求，同时对其进行重新组织以易于理解，我们就能捕获所8需功能的根本所在。我们也要确保需求描述得足够详细，以便开发人员和测试人员有足够多的信息去开始工作。这些例子可以在交付过程中代替抽象的需求，并充当需求说明、开发目标以及验收测试的检验条件。

7.6 描述非功能性需求

举例说明单独的功能需求比较直观，但是很多团队在那些交叉的或者难以使用不连续的问答11来描述的功能里苦苦挣扎。在我参与的大部分实例化需求说明工作坊里，通常至少有一个人会说对于“功能性”需求这是没有问题的，但是对于“非功能”需求而言则行不通，因为后者没有那么精确。

什么是非功能性需求

如性能、易用性或者反应时间这类特性，往往称为非功能性的，因为它们与单独的功能没有关系。通常来讲，我反对将需求按功能性、非功能性来分类，但是这不是本书要讨论的话题。很多通常称为非功能性的特性却隐含着功能性。例如，性能需求可能暗含着缓存功能、持久化约束等。据我的经验，人们最可能将功能性需求当作非功能性需求的是那些交叉的需求（例如安全性），或者是那些非离散的却可以按照比例来衡量的需求（例如性能）。**Dan North**（在与我私下交流时）指出，那些列为非功能性的需求通常说明团队还未明确地找到该功能的项目干系人。

到目前为止，我还没有看到过一个非功能性需求无法使用例子来说明的情况。即使是可用性（这可能是软件开发中最模糊不清和最主观的概念），都可以用例子来说明。请可用性专家给你看她喜欢的网站，这就是一个好的、实际的例子。这类例子的验证可能无法自动化，但是例子是实际的和精确的，足够引发一场很好的讨论。下面是一些很好的意见，可以帮助你捕获带实例的非功能性需求。

7.6.1 取得精确的性能需求

适用于：当性能是一个关键特性时

因为性能测试往往需要一个独立的环境以及与生产环境相类似的硬

件，在很多性能非常关键的系统中开发人员无法在它们的硬件上运行任何相关的测试。这并不意味着团队可以不去讨论性能需求。

→清晰地指定性能条件和举例说明可以帮助我们建立共识，并可以给开发团队提供清楚的目标实现。

在RainStor，性能对于它们的数据归档工具十分关键，所以他们会确保性能需求表达详备。他们的性能需求是以下面这种格式收集的："系统在Z个CPU上必须在Y分钟之内导入X条记录"。开发人员要么可以使用专门的测试硬件，要么让测试人员帮助他们运行测试并提供反馈。

→请记住，"要比现有系统快"不是一个好的性能需求。要告诉人们具体需要多快以及用的是何种方式。

7.6.2 为UI使用低保真度的原型

用户界面的布局和易用性无法简单地使用刚好放入真值表或自动化测试的例子来制定。不过这并不意味着我们不能讨论有关的例子。

我通常会创建一些纸质的原型，我会把用户界面元素的图样以及打印下来的网站粘贴在一起。仔细讨论一两个例子，这种方法可以很好地确保屏幕上包含了客户需要的所有信息。

商业客户通常觉得没有用户界面很难考虑问题，因为他们就是用界面进行工作的。这就是为什么当客户在屏幕上看到软件的时候经常会有"回退"发生的原因。

→有时候我们不用讨论后台的处理过程，借助于用户界面的样例

就可以预先获得更加具体的信息。

有几个我采访过的团队在使用Balsamiq Mockups[①]，它是一个提供低精度用户界面原型的网页/桌面程序。我发现纸质原型更容易使用，因为我们可以使用剪切下来的图案也可以编写备注，但是如果你想要共享工作内容，软件系统会更适合。

注释：①www.balsamiq.com/products/mockups

RainStor的Adam Knight对这种方式做了更进一步的发展，他创建了交互式原型来与客户探索模糊的需求。他说：

“我们没有使用纸质原型，而是使用**shell**脚本将几个例子做成命令行界面的原型，然后与客户一起审查这些例子，请他们详细描述他们想要如何在系统里使用新的功能。**”**

对于这种交互式工作坊提供的功能性实例，开发团队以后可以用来对需求进行说明。团队也可以使用这种方法来确定范围（详情请参阅5.2.3节）。

7.6.3 试用QUPER模型

适用于：按比例缩放的需求

当需求产生的不是离散的、精确的结果时，我们就很难去讨论它们。为什么网页需要在两秒钟之内呈现完毕，而不是三秒钟或一秒钟，你还记得最后一次对这个问题进行有意义的讨论是什么时候吗？多数情况下，我们会不经讨论或理解就接受此类需求。

在2009年的Oresund开发者大会上，Björn Regnell做了一场关于QUPER[①]的演讲，这是一个有意思的模型，用于说明那些非离散的、却可以按比例缩放的需求（例如，启动时间或者响应时间）。我还未在项目中尝试过这个模型，但是它十分有趣、耐人寻味，因此我决定将其囊括在本书中。

注释：①请参考http://oredev.org/videos/supporting-roadmapping-of-quality-requirements以及IEEE Software journal，Mar/Apr2008

QUPER使用成本、价值和质量的坐标轴来可视化按比例缩放的需求。该模型的思想是对可以按比例缩放的需求进行成本效益断点和障碍的评估，使大家可以去讨论这种需求。

QUPER模型假定这类需求会在S曲线上产生效益，而且曲线上有3个重要的点（称为断点）。可用性是产品从不可用转向可用的点。例如，手机启动时间的可用性点是一分钟。分化描述的是当某个功能发展

出会影响市场的具有竞争力的优势的时候。例如，手机启动时间的分化点是5秒钟。饱和是指质量提升变得矫枉过正时。手机启动需要半秒或者一秒对用户来说没有区别，那么一秒钟就是手机启动的一个可能的饱和点。Regnell指出超过饱和点就意味着我们在错误的地方进行了投资。

该模型的另一个假设是质量的提升并没有导致成本的线性增加。在某些点上成本会陡然增加。此时产品可能必须使用其他技术重写，或者会对架构产生显著的影响。这些点就称为该模型的成本障碍。

→为可以按比例缩放的需求定义障碍点和断点，可以让我们就产品适舍市场中的什么位置以及我们的市场定位是什么做出更有意义的讨论。

我们可以使用断点和障碍点为项目的不同阶段定义相应的目标，并且让按比例缩放的需求变得可以衡量。Regnell建议将此类需求设定成一个一个区间而非离散的点，因为持续的质量需求可以使其产生更好的效果。例如，如果你只是想让软件具备竞争软件的相同功能，那么其目标就非常接近可用性点，肯定不会超过分化点。为了让产品具有独特的卖点，其目标应该介于分化点和饱和点之间。在同一条曲线上将成本障碍可视化，将有助于项目干系人理解在不必进行超预期投资的情况下，他们可以将目标推行多远。

7.6.4 讨论时使用核查清单

适用于：交叉的关注点

通常，当客户强行增加一个全局性的通用需求时，他们会觉得更安全。我参加过很多对性能有全面要求的项目，例如，“所有页面要在一秒种内加载完成。”多数情况下，实现这样的需求（以及其他类似的全局性需求）是浪费金钱。通常来说，只有首页和一些关键功能必须在一秒种内加载完成，其他很多页面可以加载得较慢。用QUPER模型的语言来说，只有少数关键页面的加载时间需要接近于分化点，其他页面的加载时间可能更接近于可用性点。

问题是这类需求往往是在项目开始时定义的，而此时我们还不知道产品看起来会是什么样的。

Christian Hassa建议不要按照表面价值来采用这样的需求，而是要将这些交叉的需求作为讨论的核查清单。他说：

“全局性地指明整个系统‘必须在**10**毫秒内响应’很容易，但是并不是每一个功能都需要这种级别的响应时间。到底系统需要在**10**毫秒内做什么？需要发送一封电子邮件、记录动作，还是做出回应呢？我们要在心里为每个功能都创建此类非功能性条件的验收标准。**”**

→在评审一个故事的时候，为讨论准备一份核查清单可以确保你

将所有重要问题都纳入了考虑的范围。你可以用它来判定哪个交叉关注点适用于某个特定的故事，而后你就可以重点描述这些方面了。

7.6.5 建立一个参照的例子

适用于：需求无法量化的时候

由于可用性比较主观并且依赖于多种因素，所以很难将其量化。但这并不意味着无法使用例子对它进行详细说明。事实上，也只能用这种方法来详细说明。

实用性和其他类似的无法量化的功能，如可玩性和趣味性，对视频游戏来说很关键。这些特性无法简单地使用传统的用来描述需求的文档进行详细说明。Supermassive Games是一家位于英国的视频游戏工作室，他们在游戏开发中应用了敏捷过程。他们的团队使用核查清单来确保特性的不同方面都被完全覆盖，但是这样还是不足以处理那些功能的不确定性和主观性。

Harvey Wheaton是Supermassive工作室的总监，他在SPA2010会议①上的演讲里说这些功能具有“难以捉摸的特性”。据Wheaton所述，他们通常会在早期致力于将某个功能完成到最终的质量水平；然后，团队可以将其当作“完成”的一个例子：

注释：①http://gojko.net/2010/05/19/agile-in-a-start-up-games-development-studio

“我们在过程中尽早构建一个所谓的‘垂直切片’，通常是在试生产阶段的末期。这个垂直切片是游戏的一个小片段（例如，一个关卡、关卡的一部分或游戏介绍部分），并且达到最终的（可交付的）质量。通常有一个“水平切片”对其进行补充，也就是说，整个游戏的‘宽’切片，只是概略的并且低保真度的，目的是对游戏的规模和广度给出一个概念。

你可以大量使用参照的例子或概念图来描述最终产品的视觉外观和精确度，并为此聘请专门的人员制作出展示游戏外观的高质量图稿。”

Supermassive Games并没有尝试量化那些具有不可捉摸特性的功能，他们会建立一个参照的例子，这样团队成员就可以据此比较他们的工作。

→举例说明不可量化的功能，行之有效的一种方式是建立一个参照的例子。

总而言之，不要使用“非功能性需求”的分类来避免艰难的对话，要确保团队对商业用户对系统的预期（包括交叉的关注点）达成一个共

识。即使最终的例子以后不易进行自动化，我们还是有必要进行前期讨论，并使用例子来明确化和精确化用户的预期，这样可以确保交付团队专注于构建正确的产品。

7.7 铭记

始终使用同一组例子贯穿需求说明、开发以及测试阶段，以确保全体人员对需要交付的内容有同样的理解。

用于说明功能的例子必须精确、完整、真实以及容易理解。

比起在实施阶段才发现问题，真实的例子有助于更快地发现不一致的地方和功能分歧。

有了一组初始的例子后，就可以立即使用数据进行试验，同时可以寻找替代方法来测试一个功能，以便完成需求说明。

当例子很复杂并且例子太多或者出现太多因素时，你就要寻找缺失的概念并尝试使用更高层次的抽象来解释例子。使用一组集中的例子来单独说明新的概念。

第8章 提炼需求说明

“原钻是无光泽的、半透明的晶体，类似于玻璃碎片。为了加工成首饰，必须把它切割成特定的宝石形状，然后一面一面地进行抛光。**”——Edward Jay Epstein，The Diamond Invention**①

注释：①http://www.edwardjayepstein.com/diamond/chap11.htm

协作讨论是一种非常好的建立共识的方法，但是它只能推动最简单的项目，如果要推动其他项目就无能为力了。除非团队规模非常小并且项目周期也很短，可以依赖于人们的短期记忆，否则我们必须以其他某种方式来记录这些知识。

对关键实例进行讨论之后，对白板上进行拍照是获取这些知识的最简单方式。但此时的例子还很原始，原始的例子就像未经切割的钻石——非常有价值，但远未成型。从岩石中分离出真正的钻石，对它们进行抛光，并分割成易于销售的尺寸，这样才能显著提升它们的价值。对用于描述需求的关键实例来说，也是同样的道理。这些实例是非常好的着手点，但是为了使其价值最大化，我们必须对它们进行提炼和润色，以便清晰地说明重点，并创建出使团队在眼下和未来都可以使用的需求说明。

实施实例化需求说明之所以失败，一个最常见的原因是没有花时间去加工原始的例子。讨论需求说明往往需要做一些实验。我们会发现新的见解并重新组织那些实例，以便从更高的抽象层面上去看待它们。这样会带来很多好的例子，但有时也会导致陷入僵局或者产生不高明的想法。我们没有必要捕获所有过渡性的例子，也没有必要记录得出结果的过程。

另一方面，仅仅记录下那些我们想存档的关键实例而不作任何解释，是无法有效地与未参与过讨论的人一起沟通需求说明的。

成功的团队不会直接使用原始的例子，他们会从中提炼需求说明。他们会从关键实例中提取最重要的特性，将其转化成清晰明确的定义，并去掉不相干的细节，使得实现变得完整。验收条件就这样被记录下来并描述清楚，任何人在任何时候都可以拿到最终的需求说明并读懂它。这种带实例的需求说明捕获了让人满意的条件、功能的预期输出以及它的验收测试。

实例化需求说明就是验收测试

一段很好的需求说明，加上实例，实际上就是它所描述的功能的验收测试。

理想情况下，带实例的需求说明应该从业务角度出发清晰地定义所需的功能，而不是去定义系统应该如何实现细节。这样，开发团队才可以自由地找出符合需求的最佳方案。要达到这样的目标，需求说明应该满足以下条件：

精确、可测；

是真正的需求说明，而不是脚本；

是关于业务功能的，而不是关于软件设计的。

一旦完成了某个功能，描述它的需求说明就会派上其他用场。它将成为系统功能的文档，并提醒我们功能退化的情况。为了发挥它作为长期功能文档的用处，我们所编写的需求说明必须能让其他人在创建数月甚至数年之后拿起依然能很容易地理解它在做什么、为什么需要它，以及它所描述的是什么。因此，需求说明应该是：

不言而喻的；

专注的；

使用领域语言编写的。

本章侧重于讲述为实现以上目标应如何提炼需求说明。但是首先，为了从更加具体的角度去看待事情，我将展示一些好的和不好的需求说明的例子。在本章结尾，我们将运用本章给出的建议来提炼不好的需求说明。

8.1 一个好的需求说明的例子

下面是一个非常好的带实例的需求说明的例子。

8.1.1 免费送货服务

当VIP客户购买一定数量的书籍时，就为他们提供免费送货服务。免费送货服务不提供给普通客户或购买非书籍的VIP客户。

假定要获得免费送货服务，至少需要购买5本书籍，那么我们会得到下表这样的预期：

8.1.2 实例

客户类型	购物车中的物品	送货服务
VIP客户	5本书	免费，标准
VIP客户	4本书	标准
普通客户	10本书	标准
VIP客户	5台洗衣机	标准
VIP客户	5本书，1台洗衣机	标准

该需求说明不言自明。我经常会在会议上与工作坊中给人们展示这个例子，从来不必再多说一个字去解释它。标题与引言部分解释了该例子的结构，读者不需要反向从数据去理解它所说明的业务规则。其中也包含了真实的例子，这让需求说明具备了有可测性并且阐释了边界用例的行为，例如，如果人们买了10本书又将如何。

这是一个需求说明，而非一段描述人们如何测试这些例子的脚本。它并未提及程序工作流或者session约束，也未说明如何购买图书，而只是说明了送货服务的机制。它并非要讨论任何实现细节，那些都将留给开发人员以最佳的方式去实现。

该需求说明专注于免费送货服务的具体业务规则，它包含的只是与此相关的那些属性。

8.2 一个劣质需求说明的例子

与前面的需求说明相比，图8-1所示的实例是一个非常差的需求说明①。

注释：①这个实例来自于真实的项目，包含在原先的FitNesse版本中。在2010年6月于伦敦举办的一次工作坊中，我们将其作为提炼需求说明的练习。最终，FitNesse发行版本中的例子也因此做了变更。

简单的工资单验收测试

首先我们增加一些员工信息。

员　　工			
Id	姓名	地址	工资
1	Jeff Languid	10 Adamant St; Laurel MD 20707	1005.00
2	Kelp Holland	128 Baker St; Cottonmouth, IL 60066	2000.00

接下来我们给他们发放工资。

支付日期	
支付日期	支票号码
2001年1月31日	1000

我们要确保他们的工资数目正确。空白的格子由工资检查程序的测试固件（fixture）来填写，已经填入的数据将会受到检查。

工资检查程序				
Id	金额	号码	姓名	日期
1	1005			
2	2000			

最后我们要确保输出数据包含且仅仅包含2张支票信息，并且支票号码都正确无误。

工资检查程序
号码
1000
1001

图8-1 令人费解的需求说明

尽管这些表格拥有标题，周围也有一些文字，似乎解释着这一切，但实际上却没什么效果。为什么这份文档要冠以“简单”两字？显然，它与工资单有关，但它到底在说明什么？

这份文档在说明什么实际上并不清楚。我们不得不从测试数据去理解业务规则。它似乎要验证打印出来的支票必须拥有唯一号码，号码的起始数字由参数指定。它似乎还要验证每张支票上打印的数据。同时还在文字中描述了要为每位员工打印一张支票。

这份文档有许多地方容易变得很复杂。除了准备阶段(setup)，文档的其他地方并没有用到姓名和地址。表格里出现了数据库标识符(Id)，但它们与业务规则毫不相干。该实例中的数据库标识符用于在工资检查程序表格中匹配员工，这让需求说明引入了软件技术的概念。

工资检查程序的概念显然只用于测试的目的。当我第一次读到这个

实例时，我想象起Peter Sellers英国著名喜剧演员穿着Clouseau电影公司的服装检查支票的情景。我确信这并不是业务概念。

另一个有趣的地方就是需求说明断言部分的空白单元格，而且这两个工资检查程序的表格看起来似乎是没什么关联。这个实例来自FitNesse，这个工具会在空白单元格里打印出测试结果，用于故障处理，而不会做任何检查。这让该需求说明变成了大家必须仔细检查的自动化测试，几乎完全违背了自动化的目的。FitNesse的测试中有空白单元格，这通常是测试不稳定的一个迹象，同时也是测试存在问题的一个信号。要么该自动化测试与系统挂钩的地方不对，要么隐藏有潜在规则，使得测试结果变得不可重复，而且不可靠。

该需求说明中使用的语言也不统一，这使得输入和输出很难进行关联。最后一个表格里的数值1001代表什么意思？列名说这是一个号码，它确实是“号码”，但是说了等于没说，完全无厘头。第二个表格有一列支票号码，但它是何种号码呢？它们两者之间有什么关系呢？

表格里包含了地址信息，假定这么做的原因是，为了自动化包装而打印的对账单必须带有地址信息，但是基于该需求说明的测试至少无法验证一件非常重要的事情：员工们能否各自得到正确的工资金额。如果表格中的第一个人同时获得了这两张支票，这个测试会顺利通过。如果他们两个各自拿到了对方的工资，该测试也会通过。如果支票上打印的日期在很久的将来，我们的员工很可能无法兑现，但测试依然会通过。

现在我们来看看存在空白单元格的真正原因。支票没有指定顺序，这是一个功能缺失，使得系统很难按照可重复的方式进行测试。该FitNesse页面的作者决定绕开这个需求说明的技术难点，他没有在自动化层上去做这件事情，而是创建了一个可能会误报的测试。

没有更多的上下文信息，很难确定这个测试是不是只验证单个功能。如果支票打印系统还有其他用途，我更愿意把“支票号码是唯一的并起始于配置的值”这样一个事实分开到单独的FitNesse页面上。如果我们只打印工资支票，那很可能这个需求说明是工资支票打印功能的一部分。

在本章稍后，我们将对这份极不友好的文档进行提炼。但是首先，让我们回顾一下什么才是好的需求说明。

8.3 提炼需求说明时要关心什么

在本章的介绍部分，我列举了一些好的需求说明的目标。下面是一些如何实现这些目标的好点子。

8.3.1 实例要精确可测

需求说明必须是衡量成功与否的客观标准，可以明确地告诉我们何时完成了开发。它必须包含可验证的信息（输入参数与预期输出的组合），可用来对系统进行检查。

为了满足这些条件，需求说明必须基于精确的、实际的例子。关于如何确保实例的精确性，请参考7.2节。

8.3.2 脚本不是需求说明

业务人员通常考虑的是通过用户界面或几个步骤来执行某个操作，以此解释他们将如何使用系统来完成某件事情，而不是解释系统应该具有什么功能。此类实例是脚本而非需求说明。

脚本解释如何测试某个东西。它通过与系统较低层次的交互描述业务功能。脚本要求读者从操作开始向上推导，理解什么是真正重要的，以及它到底在描述什么。脚本还会把测试与工作流及session约束混合在一起，即使背后的业务规则不变，将来它们仍有可能会改变。

需求说明解释系统在做什么。它以最直接的方式去关注业务功能。需求说明较短，因为它们直接描述业务概念。这让它们比脚本易于阅读和理解。同时需求说明比脚本稳定得多，因为工作流与session约束的改变不会影响到它们。

下面是一个脚本的例子。

（1）以用户Tom进行登录。

（2）跳转到主页。

（3）搜索《实例化需求说明》。

（4）把第一个搜索结果加入到购物车中。

（5）搜索《测试之美》。

（6）将第二个搜索结果加入到购物车里。

（7）验证购物车中的物品数是否为2。

这个脚本告诉我们某件事情是如何完成的，但它并没有直接解释我们在说明什么。在阅读下一段以前，请先拿出一张白纸，试着写下这个实例到底在说明什么。事实上你能写下一些东西吗？如果可以做到，你觉得它是这个例子唯一可能描述的东西吗？

这个实例要描述的东西有太多的可能性。一种可能是购物车可以加入多个物品。另一种可能是用户登录后购物车是空的。第三种可能是《实例化需求说明》的第一个搜索结果以及《测试之美》的第二个搜索结果可以被加入到购物车中。

这是一个非常精确的并且可测试的实例，我们可以执行并确认系统是否会给出预期的结果。这个脚本的问题在于它没有包含任何信息，能用于描述它实际代表的功能。编写这个脚本的人在首次实现这个功能

时，可能非常清楚地了解它应该具有什么样的行为。但6个月之后，这点就变得不再明显了。

这个脚本不是一个很好的沟通工具。实际上我们没办法说清楚它派什么用场，也不清楚系统哪个部分有错误。如果基于这个脚本的测试突然出现失败，那就必须要有人花费许多时间去分析不同地方的代码。

脚本开始的第一个步骤，要求以用户tom来登录，这很可能是受到网站工作流的制约。除非这个实例描述的业务规则与这位特定的用户有关，否则登录用户是否是tom是无关紧要的。如果他的账号由于某种原因被禁用了，那么这个测试就会出现失败，而系统本身却不一定会有什么问题。要找到这个测试失败的原因必然会浪费很多时间。

从长远来看，使用脚本而不是需求说明来捕获验收测试会耗费很多时间，如果我们能预先用几分钟来调整那些实例的结构，那么就可以节省很多时间。关于如何把这样的脚本提炼成有用的需求说明，请参考本章结尾的8.7节。Rick Mugridge和Ward Cunningham在Fit for Developing Software一书中提供了诸多建议，可以帮助我们把脚本重组成更好的需求说明。

8.3.3 不要使用流程式的描述

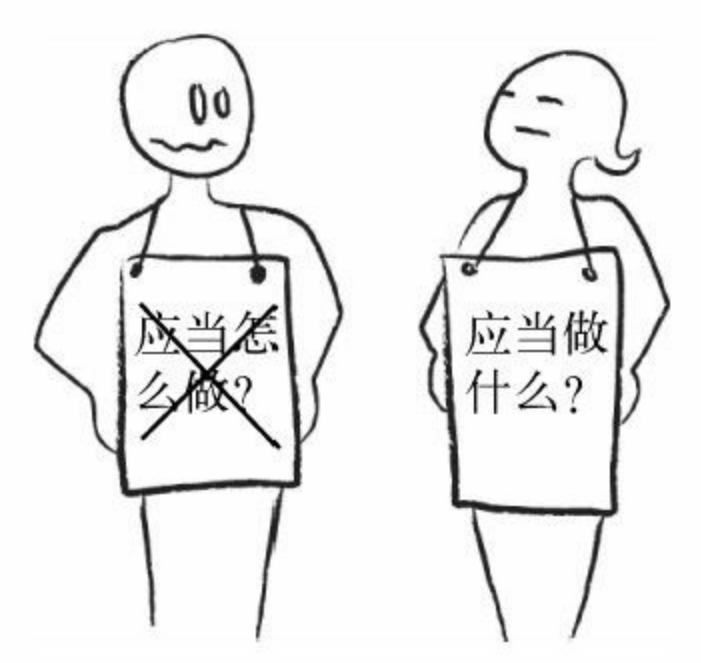
要提防流程式的描述（先做这个，然后做那个……）。除非你是在制定一种真正的处理流程，否则这通常是使用脚本来描述业务规则的信号。这样的脚本会造成许多长期维护的问题。

→要提防关于系统应该如何工作的描述。我们应当去考虑系统应该做什么。

Ian Cooper位于Beazley的团队在开始实施实例化需求说明6个月后意识到了这一点。在一次团队回顾会议上，他们开始提出自己的验收测试维护成本太高，并开始寻找降低成本的方法。后来他们把脚本重组成了需求说明。Cooper说道：

“我们用于自动化测试的模型与手工测试的模型是一样的，都把测试转化成了脚本。我们的早期测试遵循了脚本化的方法，测试就是一长串操作，加上最终的一些检查。一旦我们的思维转换到‘系统应该做什么’的模式上，事情就变得容易多了。”

把验收测试描述成脚本而不是需求说明，是团队在早期最容易犯的一个错误。相对而言，把脚本作为短迭代的开发目标来使用是比较好使的，因为大家在首次实现某个功能时，仍然会记得脚本描述的内容是什么。但想要在今后去维护并理解它们都会比较困难。这个问题可能要在数月之后才会浮现，但一旦出现，就会产生严重的危害。



8.3.4 需求说明应关注业务功能，而不是软件设计

原则上，需求说明不应该牵涉到软件设计。它应该解释业务功能，对软件如何实现不做任何规定。这样做有两个目的：

让开发人员在当下找出最佳的解决方案；

让开发人员在未来改善他们的设计。

关注于业务功能的需求说明，没有描述实现细节，这使得修改实现更加容易。当软件设计得到改善时，没有谈及任何软件设计的需求说明就无需进行修改。这样的需求说明就像一个常量，有助于未来的变更。在改善软件设计之后，我们可以马上执行基于那些需求说明的测试，而不需要对它们进行修改，以此确保以前的功能仍然可以正常运行。

8.3.5 避免编写与代码紧密耦合的需求说明

→与代码紧密耦舍、严密体现软件实现的需求说明会让测试异常脆弱。

即使测试描述的业务功能没有变动，软件设计的改动也会破坏这样的测试。那些产生脆弱测试的实例化需求说明会引入额外的维护成本，而非辅助变更。Aslak Hellesøy指出了这一点，这是他从实例化需求说明中学习到的重要一课：

“我们编写的验收测试太多了，有时候它们与我们的代码耦舍得太紧密了。虽然不如单元测试那样紧密，但仍然耦舍着。在最差的情况下，进行一次较大的重构后，需要花费**8**小时以上去更新那些测试脚本。如何在编写多少测试以及如何编写它们之间找到一个良好的平衡

呢？在这个问题上，我们学到了很多。”

→要当心需求说明中来自软件实现但并不属于业务领域的名称和概念。比如说数据库标识符**(Id)**、技术服务名或者对象类名等非直接性的领域概念，以及纯粹出于自动化目的而创造的概念。请重组这些需求说明，避免使用此类概念，这样它们会更易于理解和长期维护。

技术性测试很重要，我并不是要反对此类与软件设计紧密关联的测试。但是此类测试不应该与可执行的需求说明相混合。很多团队开始实施实例化需求说明时的一个常见错误是终止所有的技术性测试，比如说单元测试或集成测试，并期望可执行的需求说明会覆盖到系统的所有方面。可执行的需求说明会指引我们交付正确的业务功能。技术性测试可以确保我们去关注系统低层次的技术质量方面。两者我们都需要，但我们不应该把它们混合到一起。技术性测试的自动化工具比我们用来自动化可执行需求说明的工具更适合于技术性测试。它们让团队可以更加容易地维护此类测试。

8.3.6 不要在需求说明中引入技术难点的临时解决方案

适用于：处理遗留系统时

遗留系统经常会有许多技术怪癖，而且它们难以修改。用户必须绕过这些技术难点，而且区分实际的业务流程和临时的解决方法会变得十分困难。

有些团队会掉入一个陷阱，他们会把这些流程的临时解决方法包含到需求说明中。这不仅会把需求说明绑定到实现细节上，还会绑定到那些技术问题上。这样的需求说明在遗留系统中没有起到协助变更的作用。它们的维护成本很快就会变得非常昂贵。对代码的细小改动，都有可能需要几个小时的时间来更新相关的可执行需求说明。

Johannes Link曾经为一个项目工作过，为了在那个项目中运行一些基本的测试场景，必须构建200多个不同的对象。那些依赖项定义在可执行的需求说明中，而没有放到自动化层里。一年后，测试维护成本变得非常之高，以至于团队不得不回滚了他们的修改。Link说：

“修改一个功能会破坏许多测试。他们无法去实现一些新的功能，因为测试的维护成本实在太高了，但他们知道自己必须保留那些测试，以便保持较低的缺陷率。”

大多数可执行需求说明的自动化工具①会把需求说明和自动化过程分离开来（详情请看第9章开始部分的附注栏内容“其原理是什么？”）。需求说明是按照人类可阅读的形式记录的，而自动化过程是利用编程语言代码的形式记录在单独的自动化层中的。

注释：①关于自动化工具的更多内容，请访问http://specificationbyexample.com。

→把技术难题放到自动化层去处理。不要试图在测试说明中解决。

这会让你在变更和改进系统时更加容易。当我们描述和维护技术上的验证过程时，在自动化层上解决技术难题能让你充分利用到编程语言的特性和工具。程序员可以运用技术和工具来减少重复、创建可维护的代码，并轻松修改它。如果技术上的临时方案包含在自动化层里，那么当你改善技术上的设计时，需求说明就可以不受影响，而临时方案也就不再需要。

将技术上的工作流放入到自动化层里也可以让需求说明更加简短，更易于理解。而最终的需求说明则会以更高的抽象级别来解释业务概念，并侧重于对特定实例集合来说重要的那些方面（详情请见本章后面8.3.13节的内容）。

8.3.7 不要陷入到用户界面的细节里

适用于：**Web**项目

→刚开始接触实例化需求说明时，许多团队会浪费很多时间为用户界面的细节描述很多不相干的例子。他们为了过程本身去遵循实例化需求说明的实施过程，而不是为了增强对需求说明的理解。

用户界面是可见的，因此它很容易理解。我见过一些项目的团队和客户，花了几个小时去描述导航菜单。而实际上这部分的用户界面没有什么风险，这些时间应该用于讨论更加重要的功能。

Phil Cowans在Songkick公司实施实例化需求说明的时候就有过这样的经历，他认为这是大家在早期会犯的一个主要错误。

“早期的时候，我们花费了太多时间去测试用户界面的琐碎方面，因为这做起来很容易。我们没有花费足够的时间去深入边缘情况和应用程序的其他路径。测试看得见的东西很容易，但是最终还是要深入了解软件在做什么，而不是用户界面看起来怎么样。考虑用户故事以及应用程序的路径确实会有所帮助。”

不要老想着用户界面的细节，考虑用户在站点上的操作过程会更加有用。协作制定需求说明时，应当按照各部分内容对业务的重要程度来投入与此相称的时间。重要的、高风险的东西应该仔细探索而不必对那些不重要的东西定义得那么精确。

8.3.8 需求说明应该是不言自明的

当一个需求说明由于功能退化测试失败时，必须有人去做调查，了

解哪里出现了问题，并找出解决之道。可能当这种情况出现时，需求说明已经写了好多年了，当初编写的人已经不在该项目组里。因此需求说明必须具备不言自明的特质就显得十分重要了。

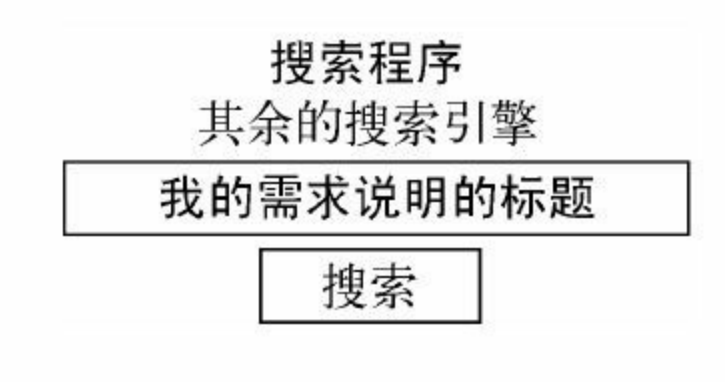
当然，确保需求说明能够不言自明，还有助于避免在首次开发其描述的功能时产生误解。

8.3.9 使用叙述性标题并使用短篇幅阐释目标

→在需求说明开始的地方节省笔墨可以产生很大的改观，还可以为今后节约许多时间。

如果需求说明只包含输入与预期的输出，那么阅读这份文档的人就不得不从实例中重建出业务规则。

为需求说明选择一个具有叙述性的标题是至关重要的。标题应该要总结出需求说明的意图。假设你在搜索Web上存放的需求说明，那么你会在谷歌的搜索框中输入的内容，就应当是你的标题。当读者搜索某个功能的解释时，这会让他们很容易找到适当的需求说明。



同时读者也要了解需求说明的结构和它的来龙去脉。请用不超过一个段落的篇幅去解释需求说明的目标以及实例的结构，并将它放在头部。撰写描述有一个很好的诀窍，就是先写例子，然后再试着向别人解释。记下你解释例子时所说的话，并将其放到需求说明的头部。

8.3.10 展示给别人看并保持沉默

适用于：独自编写需求说明时

目的：检查需求说明是否是不言自明的

→为了检查需求说明是否是不言自明的，可以让其他人来察看文档并试着理解它，而你什么都不要说。

为了确保需求说明真的能不言自明，可以请其他人来解释他们对需求的理解，看看是否符合你的意图。

当我向别人展示需求说明的时候，如果我发现自己必须给他们作解

释，我会写下这些解释并将它们放到头部。解释这些实例通常会驱使我使用更有意义的名字，或者我会插入一些注解，以便让实例更易于理解。

8.3.11 不要过度定义实例

很多团队会在他们架设好基本的自动化设施后，错误地对需求说明进行扩展，使其覆盖输入参数所有可能的组合。对这种做法，一种常见的解释是测试人员想要重用自动化框架来验证其他实例。

这种方式的问题在于它使得原有的关键实例所带来的价值被显著淡化。需求说明变得难以理解，它们不再是不言自明的。

→对于需求说明而言，定义**3**个良好的关键实例比定义**100**多个蹩脚的实例来得更加有用。

这种方式的另一个问题在于，为了验证相同的情况运行更多的实例需要更多时间，因此这会降低测试运行的速度，使交付团队获得反馈的速度变慢。

Lisa Crispin的团队在进行自动化的兼容性测试时就遭遇了这种问题，当时监管机构制定了一些规则，而这些规则并不遵循任何逻辑。Crispin和她的产品负责人一起定义了一些算法去处理许多排列组合的情况，因此，在开发工作开始之前，他们花了几个Sprint的时间编写了许多复杂的可执行需求说明。当开发人员看到这些需求说明时，他们不知所措。Crispin详细描述说：

“开发人员看到这些测试**(**可执行的需求说明**)**时，感到非常困惑，

因为这让他们‘只见树木不见森林’。他们无法使用这些测试，因为不知道要编写什么代码。因此我们发现测试应该给予我们一个宏观的蓝图，但并不需要马上给出所有细节。”

需求说明应该只列出关键的具有代表性的实例。这将有助于需求说明保持简短易懂。关键实例通常具备以下特点。

它是一个具有代表性的实例，描述了业务功能的每一个重要方面。通常商业用户、分析师或者客户会对其进行定义。

它会描述每个重要技术的边缘情况，比如技术上的边界条件。通常当开发人员担心功能分歧或者不一致性时，他们会举出这样的例子。商业用户、分析师或者客户则会对其定义正确的预期行为。

它会描述预期实现的每一个棘手之处，比如以前导致缺陷的情况，或者在以前的实例中没有描述清楚的边界条件。通常测试人员会列举出这种例子，而商业用户、分析师或者客户则会定义它的正确行为。

当然，为了关键实例重用已经实施的自动化结构，以支持那些想做更多测试的测试人员，这是有好处的。有时候我们会利用不同的边界值来探索系统行为，最简单的方法是在现有需求说明上绑定更多实例。这会使得需求说明变得太长、不够专注，而且难以理解。

不要让主需求说明变得太复杂，你可以新建一个单独的自动化测试，并在主需求说明中指出哪里可以找到它。如果你使用基于Web的活文档，那么可以使用Web链接把两个页面连接起来。如果你使用的是基

于文件的系统，那么可以在需求说明的描述中附上一个文件路径或其快捷方式。

新测试的结构可以跟原先的需求说明保持一致，并列出许多其他实例。某个功能的主需求说明仍然是有用的，它是一个沟通工具并可以提供快速反馈。为了进行广泛的测试，其他测试可以探索所有不同的组合。每次修改都可以对主需求说明进行验证，以便提供快速的反馈。而补充测试则随时可以通宵运行，以便让团队对所有额外情况建立信心。

不要试图覆盖所有情况

过度定义实例的一个常见原因是分析师或客户害怕自己由于遗漏功能而受责备。在协作制定需求说明的过程中，获取正确的需求说明是大家共同的责任，因此没有理由那么做。**André Brissette**是**Pyxis**公司**Talia**产品的总监，他指出这是他学到的一个重要教训：

“覆盖什么不覆盖什么，取决于故事成功的条件。如果你觉得那些测试真的能够覆盖成功的条件，那就没有问题。如果不是，那就有问题。在**sprint**结束之时，或者事后最终发现遗漏了什么，那么有一件事情是很清楚的：你所做的事情就是取得成功的条件，这也是大家一致认同的。这样看来，其他功能就是多余的。作为一名分析师，你不必承担责任。**”**

8.3.12 从简单的例子入手，然后逐步展开

适用于：使用许多参数组合描述业务规则

为了解决前一节中描述的问题，Crispin的团队决定在开始实现故事前只编写高层次的需求说明，详细的测试留到以后再说。

当测试人员和开发人员一起实现一个故事时，他们会制定一个容1易通过测试的路径。然后开发人员去实现自动化需求说明并编写代码，同时允许测试人员重用自动化框架并增加测试用例。然后测试人员去探索系统，使用不同的例子来做试验。如果他们找到一个失败的测试用例，就会回到程序员那里，扩展相关的需求说明并修正问题。

→不要让需求说明过于复杂，基本的例子有助于我们找到容易通过的路径，并让自动化结构落实到位。

然后，我们可以基于风险逐步尝试其他例子，并逐渐扩展需求说明。对于在一开始不易确定等价类的情况，以及边界案例由具体实现驱动的情况来说，这是一种十分有趣的解决方案。

8.3.13 需求说明要专注

一个需求说明应该单独描述一件事情：一条业务规则、一个功能，或者过程的一个步骤。专注的需求说明比那种定义多个相关规则的需求说明要容易理解。同时需求说明应该只专注于例子的关键属性，这些属性对其试图展示的内容来说必须是非常重要的。

专注对于需求说明有两个重要的好处。首先，专注的需求说明比较简短，因此相对于长的、不够专注的需求说明来说更容易理解。其次，它们还更易于维护。当系统有关的所有方面中某一方面有所变动时，一个覆盖多个规则的需求说明会受到影响，这会让基于此需求说明的自动化测试遭受更为频繁的破坏。更糟糕的是，当这样的测试失败时，很难找到问题的症结。

8.3.14 在需求说明中使用“Given-When-Then”语言

目的：让测试更容易理解

→一般来说，一个需求说明应该要声明上下文环境，指定一个单一的动作，然后定义好预期的后置条件。

可以提醒我们这么做的一个好方法是使用Given-When-Then或者Arrange-Act-Assert。Given-When-Then是定义系统行为的一种常见格式，它由早期的关于行为驱动开发的文章推广开来。它要求我们以3个部分来编写系统行为的场景：

假定(Given)一个前提；

当(When)某个行为发生时；
那么(Then)后置条件就会得到满足。
有些自动化工具，比如Cucumber[①]和SpecFlow[②]，在编写可执行的需求说明时，要求准确地使用这种语言。图8-1就是一个例子。即便是那些使用表格、基于关键字或者自由文本系统的工具，按照Given-When-Then的格式来组织需求说明也是一个非常好的想法。

注释：①http://www.cukes.info

注释：②http://specflow.org

只触发一个动作是至关重要的。这可以确保需求说明只专注于该动作。如果需求说明列举出了几个动作，那么为了理解最终的结果，读者就必须分析并理解这些动作是如何进行协作的。
如果一组动作从业务流的角度来看显得十分重要，那么给它起个名字，并把它作为更高层次的概念来使用，或许就比较重要了。这样它在领域代码中就应该以更高层次的方法来捕获，然后这个更高层次的方法就可以在需求说明中列举出来。
需求说明仍然可以定义几个前置条件和后置条件(在Given和Then部分定义多个条款)，前提是它们与测试定义的功能有直接关系。下面这个Cucumber测试的例子就有2个前置条件和2个后置条件。
Scenario：新用户，可疑交易
Given 一个用户以前从没有交易记录，
And 用户账户的注册地是英国，
When 用户发出一个运送到美国的订单，
Then 这笔交易要标识成可疑交易，
But 用户看到的订单状态是“挂起”。
使用Given-When-Then的一个潜在缺点是它像散文，它往往会吸引大家去考虑交互的流程，而不是考虑如何直接描述业务功能。请利用本章之前8.3.2节中的建议来避免此类问题。

8.3.15 不要在需求说明中明确建立所有依赖

适用于：处理复杂的依赖（或引用的完整性）时
在那些需要复杂配置的数据驱动项目中，对象一般都不能被单独创建。例如，一个支付方法的领域验证规则可能要求它必须属于某个客户，而该客户必须拥有一个有效的账户，如此等等。
→许多团队会错误地把所有先决条件的配置和设置放入到需求说明中。尽管从概念上来说，这会让需求说明清楚完整，但同时它会让

需求说明难以阅读和理解。

此外，在配置中对任何对象或属性的修改都会破坏基于这个需求说明的测试，即使它与描述的业务规则没有直接关系。

清晰地描述所有的依赖还会隐藏数据相关的问题，因此在数据驱动的项目中这是尤其危险的。

Jonas Bandi曾经参与过一个项目，为学校重写一个遗留的数据管理系统，他们遇到的一个最大问题与了解现有的数据有关。他们的团队编写了需求说明，用于动态地建立整套上下文环境。需求说明中的上下文是团队基于自己的理解定义的，并不是依照真实数据的变化来制定的。只有当他们把代码和来自于遗留系统的数据连接起来的时候，他们才能检测到需求中的许多分歧和不一致之处，而这时已经是迭代中期了（参见7.4节）。

Bekk咨询公司的团队在挪威的Dairy Herd Recording System项目中遇到了类似的问题，但他们的角度不同。他们的项目也是数据驱动的，许多对象都需要复杂的构建过程。起初，他们在每个可执行的需求说明中定义整套上下文环境。这要求大家完全猜测出所有依赖。如果遗漏了一些数据，那么即使正确地实现了代码，基于这些需求说明的测试也会因为数据完整性的约束而失败。

这些问题可以在自动化层而不是晨需求说明中得到很好的解决。把所有跟需求说明的目标不相干的依赖全部移入到自动化层中，并保持需求说明只专注于重要的属性和对象。同时请参考9.5节，那里会介绍一些不错的解决方案，可用于解决技术上的数据管理问题。

8.3.16 在自动化层中应用缺省值

→创建有效的对象是自动化层的职责。

自动化层可以使用合理的缺省值预先填充对象，并建立起相关依赖，这样我们就不必显式地对它们进行说明。这使得我们在编写需求说明的时候只关注于重要的属性，以让需求说明更容易理解和维护。

例如，在用户进行支付之前，我们无需了解创建一个客户所需的所有的地址信息，以及为其注册一张有效信用卡所需的所有信用卡属性，只需说明他的卡上有100美金即可。其他的都可以由自动化层动态构造。

这样的缺省值可以在自动化层中设定，也可以在全局的配置文件中提供，具体取决于商业用户是否可以对它们进行修改。

8.3.17 不要总是依赖缺省值

适用于：有许多属性的对象

虽然合理的缺省值让需求说明更容易编写和理解，但有些团队会过

度运用这种方式。在编程语言中消除重复通常是一种很好的做法，但对需求说明而言却不见得如此。

→如果一个实例的关键属性与自动化层提供的缺省值相匹配，尽管可以省略，但显式地对其进行说明仍然是明智的做法。

这可以确保需求说明对读者而言拥有一个完整的上下文环境，同时也让我们可以在自动化层中修改那些缺省值。Ian Cooper警告说：

“尽管一个实例的属性实际上与缺省值是一样的，但请不要依赖于它。要明确地对其进行定义，这样我们今后就可以修改那些缺省值。这也使得重要内容变得显而易见。当你读到需求说明的时候，你可以看到有些例子对这些产品有关的数值进行了说明，这时你就可以提出：‘为什么这很重要？’”

8.3.18 需求说明应使用领域语言

功能性的需求说明对用户、业务分析师、测试人员、开发人员以及任何想要了解系统的人都十分重要。为了让需求说明便于这些用户组的人员访问和阅读，需求说明必须用每个人都能理解的语言来编写。文档中使用的语言也必须一致。这会最大限度地减少翻译的需要以及误解的可能性。

统一语言(Ubiquitous language，请看下文附注栏)非常符合以上的要求。要确保你在需求说明中使用统一语言，并且当心那些似乎是出于测试目的而创建，听起来却又像是软件实现概念的类名或概念。

统一语言

软件交付团队经常会基于技术上的实现概念为项目开发他们自己的术语。这些术语不同于商业用户的术语，以致于当与商业用户进行交流时经常需要翻译。业务分析师就扮演了翻译者的角色，并成了信息的瓶颈。这两套术语之间的翻译常常会导致信息丢失，并产生误解。

Eric Evans建议发展一套通用的语言，在领域驱动的设计中作为对领域共同理解的基础，而不是去舍并不同的术语。他把这种语言称为统一语言。

如果能确保需求说明实现本节所列出的目标，我们就会得到一个很好的开发目标，而且会得到拥有长期价值、可以作为交流工具的文档。它们会支持我们演化系统并逐步建立起活文档。

8.4 提炼实战

现在让我们清理并改进一下本章前面提到的劣质的需求说明。首先，我们要给它一个漂亮的叙述性的标题，比如“薪资支票打印”，这可

以确保我们今后很容易就能找到它。同时我们应该增加一个段落解释需求说明的目标。我们确定了以下规则。

系统为每位员工打印一张支票，上面包含员工姓名、地址以及工资金额。

系统在支票上打印出支付日期。

支票号码是唯一的，从下一个可用的支票号码开始，按照从小到大的顺序打印。

支票按照员工姓名的字母顺序打印。

支票上有收款人姓名、金额以及付款日期。上面没有名字或薪资数据，那些是员工的属性。如果我们把打印的支票作为即将自动发送出去的信件的一部分，那么我们就可以说支票上还有地址信息，自动信封包装也可以使用这一信息。让我们坚持使用统一语言并一致地使用这些名词。

名字和地址的组合应该足以让我们匹配到员工的支票——我们不需要数据库标识符。

通过对订购规则（无论是什么规则）达成一致，我们可以让系统更具可测性。例如，我们可以接受按员工姓名的字母顺序来打印支票。我们可以建议客户那么做，这可以让需求说明更加有力。

为了让需求说明不言自明，让我们把上下文环境抽出来放到头部。工资日期、下一个可用的支票号码以及员工的薪资数据都是上下文环境的一部分。我们还应该清楚地说明这些数字有什么作用，这样今后大家阅读这个需求说明的时候就不必自己去弄清楚这一点了。让我们称其为“下一个可用的支票号码”。还可以将上下文环境显式地提取出来，表明它只负责准备数据而不会进行验证，这会让需求说明更容易理解。

触发动作并不一定要在需求说明中列出来。检查工资单输出结果的表格可以隐性地运行工资核算程序。这是一个例子，让我们专注于正在测试什么，而不要去关心如何进行检查。没有必要设立一个单独的步骤表明“下一步，我们向他们进行支付”。

工资检查程序是我们发明出来的概念，它违背了统一语言的规则。它在业务领域中不是一个5专有的概念，所以让我们解释一下它究竟是做什么的。因为我们想确保无论谁执行了自动化验证，都必须检查所有打印出来的支票，让我们就说“所有的支票都打印完成”好了。要不然，可能有人使用子集去匹配，也可能系统会每张支票打印两次，而我们不会注意到这些情况。

清理好的版本如图8-2所示。

薪资支票打印

系统会自动打印薪资支票：

- ❑ 每位员工一张支票，支票上有员工姓名、地址以及薪资数据；
- ❑ 使用发薪日期；
- ❑ 支票号码是唯一的；
- ❑ 下一个可用的支票号码是按升序排序的；
- ❑ 按照员工姓名的字母顺序进行打印。

▼ 薪资上下文		
发薪日期	10/10/2010	
下一个可用的支票号码	1000	
系统中的员工		
姓名	地址	工资
Jeff Languid	10 Adamant St; Laurel MD 20707	1005.00
Kelp Holland	128 Baker St; Cottonmouth, IL 60066	2000.00

在薪资程序执行期内，所有的支票都打印完成				
支票号码	日期	收款人	地址	金额
1000	10/10/2010	Jeff Languid	10 Adamant St; Laurel MD 20707	1005.00
1001	10/10/2010	Kelp Holland	128 Baker St; Cottonmouth, IL 60066	2000.00

图8-2 对图8-1显示的劣质需求说明提炼后的版本。注意它更为简短，而且是不言自明的，同时还有一个清晰的标题

与原先的需求说明相比，这个版本更加简短，并且不容易发生混乱。它要容易理解得多。在提炼完需求说明之后，我们可以尝试回答这个问题：“我们有遗漏的地方吗？”通过对输入参数进行试验，并考虑那些可能属于有效输入却违反业务规则的边缘情况，我们可以确定这个需求说明是否完整。没有必要考虑无效的员工数据，因为这应该由系统的其他部分去检查。

一种启发式的试验数据是使用数值的边界条件。例如，如果员工的工资是0会怎样？这是一个有效的用例。一个员工可能请了无薪假期或者合同暂停了，又或者他已经离职了。我们仍旧打印支票吗？如果我们遵守“一位员工一张支票”的规则，那么对于任何一位几年前就解雇了的员工，虽然不再向其支付工资，但系统仍然会打印出一张支票，只不过上面的金额是0。我们可以和业务人员进行讨论，加强一下这条规则，确保没有必要的时候就不要打印支票。

也许我们想对需求说明做进一步的提炼并将其拆分成多个需求说明，这取决于工资单是否是支票打印的唯一用例。有人可能会描述通用

的支票打印功能，比如唯一顺序的支票号码。还有人可能会描述工资单打印特有的功能，比如打印的支票数量、正确的工资金额，等等。

与工具无关

很多人会因为像图**8-1**所示那样的可执行需求说明验证失败了而抱怨**FitNesse**。诸如**concordion**之类的工具意图防止此类问题。其他工具（比如**Cucumber**）则提升了**Given-When-Then**的文本结构，以试图避免难以理解的表格。

且不要过早下定结论认定某个工具是某种问题的解决方案，要知道我同样看到过许多劣质需求说明，它们几乎都是由主流工具编写的。问题的症结不在于工具；同样，解决方案也与工具无关。问题的大部分原因是团队没有投入精力让需求说明易于理解。提炼需求说明无需多花很多精力，却可以带来更多的价值。

提炼需求说明的益处有时候并不能立马显现，因为协作可以帮助我们对预期的功能建立起共识。这就是为什么许多团队并不认为提炼需求说明很重要，最终导致出现大量难以理解的文档的原因。从关键实例开始提炼是至关重要的步骤，这可以确保我们的需求说明作为沟通工具具有长期的价值，同时它们可以为活文档系统创建良好的基础。

8.5 铭记

不要直接使用最初的实例，要对它们进行提炼，以得出需求说明。

为了充分利用实例，最终的需求说明应该是精确的、可测的、不言自明的、专注的，并以领域语言编写，同业务功能相关。

在需求说明中要避免使用脚本，避免谈及软件设计。

不要试图覆盖所有用例。需求说明不是用来替代组合回归测试的。

所有重要的用例集，都要先从一个例子开始着手，并增加值得程序员和测试人员特别关心的例子。

在需求说明、软件设计以及测试中定义并使用统一语言。

第**9**章 自动化验证而不修改需求说明

提炼好功能的需求说明之后，我们就有了实现要达到的清晰目标，并且有了一个精确的方式来衡量何时已经实现了该目标。每当系统有所变更，提炼过的需求说明就可以用来检查原有功能是否依然生效。由于举例说明的内容太过详细，我们无法在短迭代内手动执行所有检查，即使是中型项目也无法做到。一个显而易见的解决方案就是尽可能地将这些检查工作自动化。

自动化验证带实例的需求说明与传统软件项目中的测试自动化有所不同。如果我们在自动化过程中必须大量地更改需求说明，那么我们将再次陷入“传话游戏”的困境，并且还会失去提炼需求说明带来的价值。理想情况下，在自动化需求说明的验证过程中，我们不能扭曲任何信息。因此，它比传统的测试自动化更具挑战性。

本章涉及的内容包含如何自动化验证而不修改需求说明，如何控制自动化的长期维护成本，以及在我所采访的团队中引起最多问题的两个方面：用户界面的自动化与自动化测试的数据管理。文中描述的做法适用于任何工具，所以我不会针对某个工具展开讨论。如果你对这个话题感兴趣，并且想做更深入的研究，请查阅http://specificationbyexample.com并下载相关文章。在开始之前，我先回答一下邮件组与论坛里常见的一个问题：我们是否真的需要这种新型自动化？

由于自动化是技术性很强的问题，所以本章将比其他章节更偏技术性。如果你本人并非开发人员或自动化专家，那么你可能会觉得某些部分很难理解。我建议你只阅读本章前两个小节而略过其余部分，这样你也不会遗漏任何你感兴趣的内容。

其原理是什么

所有最热门的自动化可执行需求说明的工具都具有两种工件：可读形式的需求说明与编程语言中的自动化代码。需求说明的格式可以是纯文本、**HTML**或其他的可读格式，不同的工具略有区别。工具能够从需求说明中提取输入与预期的输出，然后可以将其传入自动化代码并判定结果是否与预期相符。自动化代码**(**加上一些称为夹具或步骤定义的工具**)**会调用程序的**API**、与数据库进行交互，或者通过程序的用户界面执行操作。

自动化代码依赖于需求说明，反过来需求说明并不依赖于自动化代码。这就是为什么这些工具可以让我们进行自动化验证而无需修改需求说明的原因。

有些工具需要我们将例子存放在程序代码中，并从中产生可读的需求说明。从技术上讲，效果是一样的，但是此类工具却让那些想要编写或修改需求说明而又不熟悉程序代码的人望而却步。

9.1 非得自动化吗

今天，团队在实施实例化需求说明时面对的一个最大的问题，就是可执行需求说明的长期维护成本。虽然用于自动化可执行需求说明的工具正在迅速改善，但是在易维护性以及开发工具的集成性方面，它们依

然与业已得到认可的单元测试工具相去甚远。自动化还给团队增加了额外的工作。因此它经常引起一些讨论，比如是否非得自动化不可，自动化的投入是否值得。

反对自动化的理由是它会增加软件开发与维护的工作量，并且团队对所做事情的共识来自于举例说明，而非自动化。Phil cowans指出这个观点忽视了实例化需求说明的长远利益：

“貌似为了构建一个功能你需要编写两倍的代码。但是要知道代码的行数或许并非开发过程的限制因素，所以这个观点很傻很天真。你其实没有考虑到一个不争的事实：自动化可以让你在维护已有功能或处理测试与开发之间的误解方面少花很多时间。”

对大型团队来说，自动化通常都是很重要的，因为它可以确保我们对是否完成了某样东西拥有公正客观的衡量标准。对此，Ian Cooper有一个很好的类比：

“当我疲惫的时候，刷完盘子后就不想把它们擦干。我想所有的盘子都刷好了，差不多就算做完了。但是老婆大人可不这么认为。对她来说，‘做完’意味着所有的盘子都必须是干的并且摆放好，而且水槽也是干净的。自动化就是强迫开发人员要诚实厚道。他们不能只完成自己感兴趣的那部分工作。”

就长期而言，自动化也很重要，因为它让我们可以更加频繁地检查更多的用例。Pierre Veragen说他们公司的经理在很短的时间内就理解了它的价值所在：

“突然之间，经理们意识到在一个测试里检查的不只是两三个数字，我们现在可以检查**20**或**30**个乃至更多的数字，而且还可以更容易地找出问题所在。”

有些团队通过转向技术工具来降低自动化成本。当我在为本书而做采访的时候，有件事情令我感到很惊讶：Jim Shore是敏捷社区的一位思想领袖，并且还是实例化需求说明的早期采用者，但由于成本他实际上放弃了对可执行需求说明进行自动化。[①]Shore写道，根据他的经验，比起自动化验证而不修改需求说明，举例说明可带来的价值更多：

注释：①我们的部分email往来公布在网上。请看http://jamesshore.com/Blog/Alternatives-to-Acceptance-Testing.html、http://james-shore.com/Blog/The-Problems-WithAcceptance-Testing.html以及http://gojko.net/2010/03/01/are-tools-necessary-for-acceptance-testing-or-are-they-just-evil/。在这些文章的链接里，你也可以找到社区其他成员的看法。我强烈建议大家阅读这些文章，特别是Shore关于自动化可执行需求说明替代方案的讨论。

“我使用**FIT**和其他敏捷验收测试工具的体会是这样的，它们的成本大于所带来的价值。从实际的客户和业务专家那里收集具体的例子具有很大的价值，而使用诸如**FIT**之类的‘自然语言’工具并不能带来多少价值。”

根据我的经验，这种做法在短期内可以节省时间，但它却使团队无法从实例化需求说明中获得长远的利益。

在决定是使用一种技术工具来自动化需求说明的验证还是使用可执行需求说明的自动化工具时，请考虑一下你想要从中获得哪种好处。如果我们使用一种技术工具来自动化那些例子，那么自动化会更加容易并且维护成本更低，但却失去了今后用它们与商业用户进行沟通的能力。我们可以获得很好的回归测试，但是这些需求说明只有写这些测试的人才能看得懂。根据你所处的环境，你需要判断这是否可以接受。

自动化验证而不修改需求说明是建立活文档的关键部分。舍弃这部分，我们就不能保证可读的需求说明的正确性。对于许多团队来说，实例化需求说明的长远利益来自于活文档。因此，不要舍弃自动化，它们会保留着原始的需求说明，我们转而需要想办法去控制维护成本。在本章稍后的9.3节及本书第10章，你将看到很多团队用来降低长期维护成本的好方法。

9.2 从自动化开始

可执行需求说明的自动化验证完全不同于开发人员和测试人员常用的单元测试、录制与脚本化的功能自动化。自动化的同时保留可读性，需要团队学习如何使用新的工具并探索将自动化契合到系统中的最好方式。以下是关于如何开始实施自动化过程的一些好想法，以及我采访的团队在实施自动化的过程中所犯的常见错误。

9.2.1 为了学习工具，先尝试一个简单的项目

适用于：在遗留系统上工作时

一些团队会利用一个简单的项目或一个spike来学习如何使用新的自动化工具。如果你的工作管道中有相对独立的一小部分工作，那么正好可以使用这种方法。

→小的项目可以最小化风险，有助于你专注地学习如何使用工具，而无需处理复杂的集成与业务规则。

如果你在转型到敏捷开发过程的同时想要实施实例化需求说明，这种做法将会特别有效。

在uSwitch，当他们准备引入Cucumber（可执行需求说明的另外一种热门自动化工具）时，他们就是采用了这种方法。他们让整个开发团队将原有的测试转换到新的工具上，这使得团队的每个人都迅速体验了一下新的工具。Stephen Lloyd说同时这也展示了可执行需求说明的威力：

“我们意识到还有一整套其他级别的测试需要完成，并且当前开发周期尾声的测试没有太多意义。**”**

利用小型项目可以学习并锻炼新的技能，而不会对正在进行的开发工作带来多少风险，因此相较于其他风险更高的试验方法，这种方法更容易获得批准。

将结果交给外部的咨询师来评审或许是个不错的主意。做完一部分可以评审的内容后，外部咨询师能够提供更有意义的反馈并讨论出更好的方案。到那时，团队也将获得试用工具并巩固基础知识的机会，这样他们就能理解更先进的技术并从咨询师那里获得更多的价值。

9.2.2 事先计划自动化

在那些没有事先设计好自动化测试的系统上工作时，他们在自动化可执行需求说明的初期，生产力会下降。

即使不考虑学习使用新工具所带来的消耗，自动化验证在初期还是会给项目增加显著的开销。自动化需要有前期准备，开始自动化之前有大量的工作必须完成，包括创建基础的自动化组件、确定可执行需求说明的最佳格式及与系统集成的最佳方式、解决测试稳定性与专用环境的问题以及其他诸多事项。大多数此类问题我们将在本章与第10章涉及，但现在重要的是需要了解生产效率在自动化初期会有所下降。

一旦解决了这些问题并且可执行需求说明的结构稳定之后，我们就可以在新的需求说明里重用自动化的基础组件。当项目成熟后，自动化所需的工作量将会显著降低，而生产效率将极速提升。

在有些项目中，开发人员在预估实现一个故事所需的工作量时，并没有考虑到上面提到的这个问题。因而在开始进行自动化时，他们突然发现花费在自动化可执行需求说明上的时间可能就要比在生产代码中实现相关功能所需的时间多得多。

→确保事先为生产率降低做好了计划。为了在当前迭代内完成自动化，团队必须减少交付内容。

除非做过此类计划，否则自动化将会蔓延到下一轮迭代并且中断流程。这是Pyxis技术公司Talia产品的总监André Brissette得到的一个重要教训：

“如果可以重新来过，我会在一开始就更明确地指出编写测试（可执行需求说明）的必要性。我当时知道对团队来说编写这种测试是一种挑战，所以我也是很有耐心。但我本应该在**sprint**（迭代）里为编写测试腾出更多时间。当时开始之后，我们谈到可执行需求说明时，团队说：‘我们真的没有时间来跨过这道学习门槛，因为我们这个**sprint**的工作已经排满了。’事实上，他们工作负载大的一个原因是我在该**sprint**里放入了大量需要完成的功能。因此，自动化启动起来很慢，并且经过了很多轮迭代才有了一组像样的需求说明。

如果一开始就攻破这道屏障，并在初期只实现较少的功能，或许会更加有效。下一次我会选择这么做。如果你长期采取这种混杂的方式，你得到的只有成本增加而不会有所收益，代价反而更加昂贵。**”**

为了确保在计划中包含启动自动化的工作量，有一个方法是将自动化工具库当作单独的产品来开发，使其拥有自己的功能清单，然后团队把一定比例的时间用在该产品上。为了清晰起见，主要产品和自动化框架应该由同一个团队来开发和交付，这样团队就能熟悉此后要做的自动化。我建议完全将它看作是一个单独的产品，以便减少对主要交付工作的影响。

9.2.3 不要拖延自动化工作或将其委派他人

由于自动化所需的开销，有些团队推迟了自动化。他们使用例子描述了需求说明，然后编写了代码，却将自动化推迟进行。那些开发团队和测试自动化团队是独立的项目，或者自动化测试是由外部咨询师编写的项目似乎更容易出现这种情况。无论如何，这会导致大量的返工与来回折腾。

开发人员把用户故事标识为已完成，却没有一个客观的自动化标准与其相对应。最终将验收测试自动化时，往往暴露出不少问题，这些用户故事也不得不再回到开发人员手里进行修复。

当自动化与开发同时进行时，开发人员必须将系统设计得容易测试。而当自动化委派给测试人员或外部咨询师时，开发人员就不会去关心如何让系统容易被验证。这使得自动化更难、花费更高。当自动化出现问题时，还会导致测试拖延到下一个迭代并打乱流程。

→不要因为可执行需求说明的开销问题而推迟自动化，只有积极加以处理，后续的事情才会轻松一些。

推迟自动化仅仅是个权宜之计。对于早期的开发进度来说，它可能会加快交付用户故事的速度，但是那些用户故事今后还是会再次回来要求修复的。对此，David Evans经常用公交车做比喻：如果公交车不用为了上客而停车，那么它可以开得快很多；但这么一来，它就根本不能发挥作用了。

9.2.4 避免根据原有的手动测试脚本进行自动化

开始时根据原有的手动测试脚本来创建可执行的需求说明貌似是合乎情理的做法。此类脚本已经描述了系统的行为，并且测试人员也在使用，所以用来进行自动化必定会有所帮助。但真是这样的吗？事实并非如此，这恰恰是最常见的一种失败模式。

手动检验与自动化校验的约束条件是完全不一样的。在手动测试中，准备上下文环境所花的时间往往是主要的瓶颈所在。而在自动化测试中，时间主要花在了寻找测试失败的原因上。

例如，为了准备测试用户账户管理的逻辑，测试人员可能必须先登录到管理端程序，创建一个用户，然后用新创建的用户登录客户端程序并更改其密码。为了避免在测试中重复以上步骤，测试人员会在多个手动测试脚本中重用这样的测试上下文环境。这样只需创建一个用户，就可以进行多个测试，比如屏蔽该账户并验证用户无法登录，而后重置密码并验证相应功能，再然后更改用户偏好设置并验证其首页的变化。这种方式有助于测试人员更快地执行手动测试脚本。

而在自动化测试中，创建并设定用户所花的时间不再是个问题。通常，自动化测试可以比手动测试执行更多用例。当自动化测试运行正常

时，没人会去查看它们。一旦测试出现失败，则必须有人去找出问题所在。如果测试描述的是一系列相互依赖的步骤，那么由于脚本中的上下文在不停地变化，我们会很难理解究竟是什么导致了问题。

相对于较小并且更专注的测试，在单一脚本中检查10件不同的事情比较容易出错，因为不同领域的代码会影响它的结果。在之前的用户账户管理的例子中，如果密码重置功能有问题，那么设置用户偏好也将无法进行。结果导致检查用户首页变化的验证也将失败。如果我们不去使用一个较大的脚本，转而用10个较小的、专注的并且独立的测试，那么密码重置功能中的Bug不会影响用户偏好的测试结果。这让测试更具弹性并且可以降低维护成本，同时也有助于更快地找出问题。

→不要直接将手动测试脚本自动化，要考虑脚本的测试目的，并使用一组独立的、专注的测试来描述测试目的，这会显著降低自动化的开销与维护成本。

9.2.5 通过用户界面测试赢得信任

适用于：团队成员怀疑可执行的需求说明时

许多用于自动化可执行需求说明的工具允许我们在用户界面层之下与软件进行整合。这可以降低维护成本、让实现自动化变得更加容易，并且可以提供快速反馈（请参考本章后面9.3.6节的内容）。

但是商业用户与测试人员一开始可能不会信任此类自动化测试。他们在屏幕上看不到变化，就不会相信代码正在被验证。

→刚开始使用实例化需求说明时，如果团队成员质疑自动化的必要性，那么你们可以尝试通过用户界面来执行需求说明。请注意，不要去修改需求说明以描述用户界面的交互，但是你可以在自动化层里隐藏这些动作。

在挪威的奶牛记录系统项目中，获得商业用户对可执行需求说明的信任是一个艰巨的任务。来自Bekk咨询公司的经理Børge Lotre参与了这个项目，他说随着可执行需求说明中检验的增多，他们逐渐地赢得了信任：

“他们（商业用户）以前坚持要求除了**Cucumber**测试，还需要有手动测试。我认为他们正逐渐看到**Cucumber**测试的价值，因为他们无法在我们每次增加新的功能时都执行（手动）测试来验证已有的功能。**”**

因为用户界面自动化会减慢反馈速度，并且会显著增加自动化层的复杂度，所以通过界面自动化可执行需求说明通常应该是不得已而为之的。反过来说，通过用户界面执行自动化的需求说明可能是刚开始获得非技术性用户信任的好方法。要让自动化层具有一定的灵活性，这样以

后可以切换到应用程序的界面层之下与系统进行集成。

当遗留系统没有清晰地集成API时（这种情况下，自动化测试的唯一方法是端到端，从前端用户界面开始，在数据库中或再次利用用户界面来验证结果），那么通过用户界面来运行可执行的需求说明也是一个不错的选择。在这种情况下，让自动化层具有一定的弹性同样是一个不错的主意，因为一旦架构变得更具可测性，你很可能会将它们转移到用户界面层之下进行自动化。

除了赢得信任，有时候在自动化测试过程中看到应用程序界面，也会有助于大家想到一些其他的例子。

根据本人的经验以及为编写本书所做的案例分析，通过用户界面执行测试不易扩展。在赢得利益相关者的信任之后，你可能会想要减少通过用户界面执行的测试。

如果你决定要通过用户界面来自动化需求说明，那么请应用本章后面9.4节所描述的想法，以便充分利用其价值并确保在必要时可以将自动化测试转移到用户界面层之下。

9.3 管理自动化层

我采访的那些团队长期面对的一个最大挑战是控制活文档系统的维护成本。其中一个主要的因素是有效地管理自动化。

本节中，我将介绍一些团队用来降低自动化层长期维护成本的一些好想法。本节的建议适用于各种自动化工具。

9.3.1 别把自动化代码当作二等公民

团队经常会犯的错误是，认为需求说明或相关的自动化代码没有生产代码重要。将编写自动化测试的任务交给能力较差的开发人员与测试人员就是证据，同时也没有投入像生产代码那么多的精力来维护自动化层。

很多情况下，这种认识来自于一种误解，那就是实例化需求说明仅仅是功能测试的自动化（也就是敏捷验收测试和验收测试驱动开发），同时开发人员认为测试代码没那么重要。

Wes Williams说这使他想起了早期使用单元测试工具的经验：

“我想这与编写**JUnit**有类似的学习曲线。一开始我们对**JUnit**测试采取了同样的态度，接下来每个人都参与到测试编写中，**‘**嗨！伙计们，**JUnit**也是代码，也应该要整洁清晰**’**。如果不这么做就会碰到维护性问题。后来我们了解到测试页面（可执行的需求说明）本身也是**‘**代码**’**。**”**

Phil cowans将这个问题列为他的团队早期在Songkick实施实例化需求说明时所犯的一个最大的错误。他说：

“测试代码与应用程序的常规代码一样也是一等公民，也需要维护。我现在甚至认为（验收）测试是一等的，而（生产）代码本身则不如它重要，因为测试是对应用程序能做什么事情的权威描述。

归根结底，要取得成功，更多的是要解决如何构建正确的东西，而不是正确地构建。如果测试描述的是代码具有的功能，那么它不仅仅是开发过程中非常重要的一部分，而且它对于构建产品、理解构建内容并保持复杂度可控也是非常重要的一部分。当时我们大概花了一年时间才意识到这一点。”

Clare McLennan指出，让最有能力的人去设计和构建自动化层是非常关键的：

“有一天我回去后，某个开发人员说测试集成框架的设计几乎比实际产品的设计还重要。也就是说，测试框架需要拥有与实际产品一样好的设计，因为它必须是可维护的。测试系统之所以能成功的部分原因是，我们了解到它的结构并且可以读懂它的代码。

项目中他们往往会让初级程序员编写测试代码与测试系统。然而，自动化的测试系统是很难设计好的。初级程序员往往会选择错误的方法，而且他们构建的东西也不大可靠。所以请用最好的架构师来做这件事情。他们要有能力说出这样的话：如果我们要在设计中修改

这个，那么它会好很多并且更容易测试。”

我还不至于认为自动化代码比生产代码重要。归根结底，构建软件的目的是因生产代码可以帮助我们达成一些商业目标。如果没有很好的生产代码，那么即使是世界上最好的自动化框架也无法让项目取得成功。

→带实例的需求说明（最终以活文档形式存在）比生产代码存活得久得多。当我们使用更好的技术完全重写生产代码时，良好的活文档系统是至关重要的。它比任何代码都经久耐衰。

9.3.2 在自动化层里描述验证过程

自动化可执行需求说明的工具大多使用纯文本或HTML格式来处理需求说明。这让我们在更改需求说明时无需重新编译或重新部署程序代码。另一方面，自动化层是程序代码，修改后需要重新编译与重新部署。

为了避免频繁修改，许多团队尝试让自动化层具有通用性。他们在自动化层中只创建低层次级别的可重用组件，比如UI自动化命令，然后使用这些命令将验证过程脚本化，比如用这些命令来验证网站的工作流。这种做法的一个明显标志是，需求说明中包含了用户界面的概念（比如点击链接或者打开窗口），或者更有甚者，在需求说明中包含了更低级别的自动化命令(例如Selenium)的操作。

例如，Ultimate软件公司的Global Talent Management团队决定在某个时候将所有工作流剥离自动化层并将其放入到测试需求说明中。他们使用了自己定制的开源UI自动化工具SWAT，以便将所有SWAT命令直接暴露为fixture。他们将SWAT命令组成具有意义的领域工作流供需求说明使用。这种方式最初使得编写需求说明更为简单，但后来导致了许多维护问题，Scott Berger和Maykel Suarez说：

“有一个中心团队专门维护**SWAT**并编写宏，到了某个时刻已经无法维护了。我们使用了基于宏的宏，这样导致了难以重构（测试），所有一切都成了一场噩梦。一个**given**（测试上下文）就是一个可折叠区域，展开后又有不计其数的内容。最后我们转而在**fixture**中实现工作流。针对每一页（需求说明）背后都有一个**fixture**与之相对应。”

→不要在需求说明里描述验证过程，而应该将它们放到自动化层里去记录。最终的需求说明会更加专注并且更容易理解。

在自动化层里描述验证过程（相对于测试“什么”而言，也就是“如何”测试）会让该层更复杂并且更难维护，而IDE之类的编程工具可以使这项任务简单化。当Berger的团队在纯文本的需求说明中将工作流描述为可重用的组件时，虽然他们使用的是纯文本编程，却没有任何开发工

具的支持。

我们可以使用编程工具来维护验证过程的代码实现，这比维护纯文本的描述更加有效。我们还可以更容易地在其他相关的需求说明中重用自动化验证过程。关于这方面的更多详细内容，请参考本章9.4.3节附注栏中的内容。

9.3.3 不要在测试自动化层里复制业务逻辑

→在自动化层里模仿应用程序的部分业务流或逻辑可以使得测试更容易自动化，但是它会让自动化层更加复杂化并且更难维护。更糟糕的是，它会让测试结果不太可靠。

如果实际的产品流程具有某个问题，而这个问题并没有复制到自动化层的流程副本当中。那么依赖于该流程的某个实例，在真实系统上执行时将会失败，而自动化的测试却会执行通过。这无疑会给团队一种虚假的保证，让大家以为一切正常。

对于爱荷华州助学贷款公司的Tim Andersen来说，这是一个最重要的早期经验：

“我们并不使用测试辅助代码创建一个‘伪贷款’，而是更改了测试代码以利用应用程序来建立一个有效状态的贷款。后来，我们的测试抽象层使用虚构人物来调用应用程序，我们得以删除了近三分之一的（自动化层）测试代码。这里得到的经验是不要伪造状态，虚幻的状态容易导致缺陷并且维护成本更高。当我们使用真实的系统来创建状态后，大量的测试失败了。我们仔细做了检查，发现使用了这种新的方法后原有的测试暴露了系统的一些缺陷。”

对于遗留系统，在自动化测试中使用生产代码有时候会导致非常严重的问题。例如，我的一个客户扩展了一个第三方的产品，那个产品的业务逻辑混合了用户界面的代码，让人无从下手。我的客户对第三方组件的代码只有只读权限。有人起初直接将第三方程序的部分代码复制到了测试fixture中，并且删除了所有对用户界面的调用。结果当第三方程序提供商更改了他们的代码之后，问题就出现了。

我重写了这些fixture，初始化第三方程序的窗口类并使用反射调用私有变量以便跑通真实的业务工作流。我从未在生产代码中做过类似的事情，这只是两害相权取其轻。我们删除了90%的fixture代码，在第三方程序提供商更改私有变量的用法后我们也必须偶尔修改自动化测试，但是这种做法的工作量远远少于那种无时无刻都要复制并修改大量代码的做法，而且还让测试具备了可靠性。

9.3.4 沿着系统边界自动化

适用于：复杂的集成环境

→如果你工作于一个错综复杂的系统，那么理解自己的职责边界所在是很重要的。请沿着这些边界说明需求并自动化测试。

对于错综复杂的系统来说，可能很难甚至无法在一个自动化测试里包含整个端到端的流程。当我采访rob Park时，他的团队正在集成一个将语音转换成数据的外部系统。要让每个自动化用例都走完整的流程，即使有可能，也是不切实际的。他们并不是在开发语音识别程序，只是与这样的系统进行集成而已。

他们的职责在于在语音信息转换成数据后对其进行处理。Park说他们决定隔离这个系统并提供一个替代的输入途径以便更容易进行自动化：

“我们正在为互动式语音应答（**IVR**）编写一个功能。策略编号与鉴定信息会自动从**IVR**系统传输到该应用程序中，因此界面上会带有预填充值。在进行了第一次**“**神勇三剑客**”**会议后，结论很明显，那就是我们需要一个测试页面来准备**IVR**发出的数据。**”**

Park的团队并没有把外部系统包含进去，将那些例子进行端到端的自动化，他们将外部输入从系统中解耦出来，并且将系统中他们负责的那部分验证进行了自动化。这让他们能够使用可执行的需求说明来验证所有重要的业务规则。

商业用户很自然地会考虑端到端的验收。他们对没有包含外部系统的自动化测试感到没有信心。这个问题应该交给单独的技术性集成测试

去处理。在这个案例中，播放一个简单的预录制信息并验证它是否完整经过系统就是诀窍所在。这个测试将会验证所有组件相互之间是否能够正确地对话。因为所有业务规则都单独说明并测试了，所以我们无需为所有重要的用例都运行高层次的集成测试。

关于如何处理大型的、复杂的基础设施还有更多技巧，请参考下一章。

9.3.5 不要通过用户界面检查业务逻辑

传统的测试自动化工具主要是对用户界面上的对象进行操纵。大多数可执行需求说明的自动化工具则可以在用户界面之下直接与应用程序编程接口(API)交互。

→除非通过用户界面端到端地运行某个功能的自动化需求说明是信任该功能的唯一途径，否则请不要这么做。

用户界面自动化通常比在服务层或API层级的自动化更慢并且维护成本更高。通过自动化可视的用户界面来赢得信任（如本章前面所述）是一个特例，只要条件允许，在用户界面层之下验证业务逻辑往往是更好的方案。

9.3.6 在应用程序的表皮之下进行自动化

适用于：检查**session**约束与工作流约束

通常工作流与session的规则只能通过用户界面层来做检查，但这并不意味着自动化此类检查的唯一途径就是打开浏览器。有几个开发Web应用程序的团队没有通过浏览器来自动化需求说明，他们直接利用HTTP层（就在应用程序表皮之下）进行自动化，这节省了大量的时间和精力。Tim Andersen是这么诠释这种方法的：

“我们会发送一个很像**HTTP**请求的哈希表。它具有一些默认值，我们会改写其中某些对测试比较重要的数据，测试所做的事情与**HTTP**请求所做的大同小异。这就是我们的虚拟人物［**fixture]**的工作原理：使用对象来发送**HTTP**请求。它们就是这样使用真实的状态和对象的。**”**

不去运行浏览器，可以让自动化的检验并行执行而且运行得更快。而Christian Hassa使用了一种类似的但更底层的方法：调用应用程序内部的Web控制器。这样不仅避免了HTTP调用而且还加快了反馈速度。他是这么解释的：

“我们将一部分（需求说明）通过**Selenium**直接绑定在**UI**上，而将其余部分直接绑定到**MVC**的控制器上。直接绑定到**UI**的开销是很大的，而且我觉得这也不是这种技术的主要价值所在。如果让我选择是将需求说明全部绑定到控制器，还是将有限的一部分绑定到**UI**，我一

定会选择前者。是否绑定到**UI**对我而言是个可选项，不把所有与系统有关的需求说明都绑定到控制器上则不是个可选项，况且绑定到**UI**的成本高得多。**”**

→在应用程序的表皮之下进行自动化是一种很好的方法，这样既可以重用实际的业务流程，也可以避免在自动化层里出现重复。通过直接调用**HTTP**而不是通过浏览器来执行检查，可以让验证速度显著加快并且使执行并行检查成为可能。

浏览器自动化程序库通常很慢而且会锁定用户配置，所以一台机器上只能同时运行一个此类检查。直接调用HTTP的自动化工具和程序库有很多，比如WebRat①、Twill②以及Selenium 2.0 HtmlUnit Driver③。很多新颖的MVC框架支持在HTTP层之下进行自动化，这可以让此类检查更加高效。这些工具允许我们并行执行测试，因为它们动用的部件比浏览器自动化要少，所以更快而且更可靠。

注释：①http://wiki.github.com/brynary/webrat

注释：②http://twill.idyll.org

注释：③ http://seleniumhq.org/docs/03_webdriver.html#htmlunit-driver

选择自动化哪些东西

在**Bridging the communication Gap**一书中，我建议自动化所有需求说明。在准备此书的过程中，当我与许多不同的团队进行交谈后，我了解到有些情况下自动化是不值得的。**Gaspar Nagy**给了我两个很好

的例子：

“与验收条件所带来的利益相比较，有时自动化成本会过高，例如，将内容显示在一个可排序的表格内。用户界面控件**(widget)**本身就支持排序功能。为了检查数据是否真的被排序了，你需要大量边界案例的测试数据。这最好还是留给比较快速的手动检查去做。

我们的应用程序还需要离线功能。非常特殊的离线边界案例可能会很难自动化，同时手动测试或许已经足够好了。”

在以上两个例子中，快速的手动检查就可以让团队对系统产生一定的信心：我们的系统是用户可以接受的。自动化所需的时间会多于长期来看节省的时间。

多数情况下，对检查布局的实例进行自动化并非一个好的做法。这样做技术上是可行的，但是这对许多团队来说都会是高投入低产出的。对参照实用性的例子进行自动化（比如**7.6.5**节所建议的）几乎是不可能的。实用性和趣味性需要肉眼与主观的衡量。其他不值得自动化检查的例子有：直观性的东西，或者去判定某些东西看起来有多好或者用起来有多容易。这并不意味着对此类例子进行讨论、举例说明，或者将其存储在需求说明系统中就没有好处了，事实恰恰相反。讨论此类例子可以确保每个人都对相关功能拥有共同的认识，只是手动检查可以更为高效。

对于这些功能的测试应尽量予以自动化，这样可以帮助我们只针对那些自动化初期或长期维护成本高昂的一些少数情况，集中进行手动检查。

虽然在谈到用户界面时我主要使用了Web应用程序作为例子，但是这些建议同样适用于其他类型的用户界面。在应用程序的表皮之下进行自动化可以让我们验证工作流和session约束，而且与通过用户界面运行测试相比较，这种做法仍能缩短反馈时间。前面我们大致看了一下自动化的管理，是时候谈谈自动化的两个特定方面了：用户界面和数据管理。这两个方面导致许多团队的自动化出现了问题。

9.4 对用户界面进行自动化

对于本书调研所涉及的那些团队来讲，在实施实例化需求说明的过程中，对用户界面进行自动化是最具挑战性的方面。几乎所有我采访过的团队都在早期犯过同样的错误。他们在制定测试的时候，想要通过用户界面将测试作为一系列技术步骤进行自动化，并且往往会在需求说明中直接使用用户界面自动化的命令。

用户界面的自动化程序库使用的是屏幕对象的语言，本质上是软件

设计。用这种语言描述需求说明直接与提炼需求说明的关键思想相矛盾（请参考8.3.2节与8.3.4节）。这种做法除了让需求说明难以理解外，还会使得自动化测试的长期维护异常艰难。Pierre Veragen的团队就曾因为用户界面的一个小改动，不得不抛弃所有的测试：

“用户界面测试是任务导向（点击、指向）而非行为导向的，因此它们与**GUI**的实现紧密耦舍。测试中有大量的重复。**FitNesse**测试是按照**UI**被设定的方式来组织的。当**UI**改变时，所有这些测试都必须更新。这时从概念到技术的转换发生了变化。对**GUI**的一个小改动，比如添加一个**ribbon**控件，都会破坏所有测试。我们没有办法去更新测试。”

到此为止对测试的所有投入都浪费了，因为对他们来说抛弃所有测试比更新测试来得简单。团队决定对应用程序的架构进行重组以便更容易地进行测试。

如果你决定通过用户界面来自动化验证一部分需求说明，那么对你的团队来说，有效地管理该自动化层将会是一个非常重要的行为。以下是一些很好的想法，有助于你在通过用户界面进行自动化测试的同时，保持那些测试的易维护性。

9.4.1 以更高层次的抽象来详细说明用户界面的功能

将从业务语言到用户界面对象语言的转译放入到自动化层，有助于避免长期的维护问题。本质上而言，这意味着我们要以更高层次的抽象来制定用户界面测试。Aslak Hellesøy说这就是他早先获得的一条重要经验：

“我们意识到如果以更高的层次来编写测试，我们将会收益颇丰。这让我们在更改实现时无需改变大量功能的测试脚本。因为测试更加短小，所以更容易阅读。我们有几百个这样的测试，只要瞥一眼就可以知道它们的目的是什么。而且这些测试也更具有弹性，方便修改。”

Lance Walton也有过同样的经历，最后他们在集成层里创建了一些类，这些类代表了用户界面的操作，然后他们将抽象级别提升到工作流的高度，最终又提升到更高级别的行为。他说：

“我们检查了许多已知的测试，它们按照‘键入这个，点击这个按钮’的风格编写，结果发现它们之间有大量的重复。我们本能地进行了重构并且意识到我们需要某些东西来代表屏幕（页面）。我使用了早期的**XP**规则：如果一小段表达式拥有具体的含义，那就将其重构成一个方法并给它命名。可以预知我们的每个测试都需要登录功能，它应该是可以重用的。我并不知道应该怎么做，但是我知道我们迟早会那么做的。所以我们想到了将屏幕（页面）封装成类。

接下来我们意识到，我们始终以同样的顺序来使用页面——其实就是一个工作流。之后，我们明白了工作流仍然与我们的设计方案有关，所以实际上我们可以撇开工作流进而关注用户想要完成的事情。

所以我们拥有了包含具体细节的页面，然后在此之上产生了任务，之后又形成了整个工作流，最后树立了用户想要达成的目标。当我们到达那个层次后，编写测试就很快了，并且即使程序有所改动，测试仍然会比较健壮。**”**

重新组织自动化层（以处理各种行为，同时关注于需求说明的测试，而非脚本，这样会显著减少自动化测试的维护成本）。Walton说：

“早先，想看任何内容都必须登录。后来产生了一种观念，无需登录也可以看很多内容，只有当你点击链接后才会要求你登录。如果你有大量的测试需要在一开始就进行登录，那么你的第一个问题就是，在你去掉登录步骤之前，所有这些测试统统都会失败。但是当你点击

某个链接之后却必须登录，所以一大堆测试也会因此而失败。如果你把登录抽象出来，那么在测试里以某个具体用户登录并不意味着立刻就执行——你可以将用户信息保存起来，当系统要求登录时再派上用场。

测试顺利通过。当然，你需要额外的测试来检查何时需要登录，但是这就另当别论了。那些用于检测用户能否实现其目标的测试，即便系统逻辑发生显著变化，它们依然还是很健壮的。我们居然可以如此容易地进行这个改动，这令我很惊奇同时也印象深刻。我真的开始看到我们有控制这些东西的能力。”

通过将用户必须事先登录才能执行某个具体操作的事实与实际行为（如填写登录表格、提交并登入）区分开来，而让自动化层决定何时在工作流中执行该操作（以及是否需要执行该操作），这样可以让基于此需求说明的测试更具修改的弹性。同时这还提升了用户界面操作的抽象级别，让读者更容易理解整个需求说明。

→从更高的抽象层次来详细说明用户界面的功能，可以让团队避免在业务概念与用户界面概念之间进行转译。同时还会让验收测试更容易理解并且更具修改的弹性，降低长期的维护成本。

关于如何组织用户界面自动化测试才能不丢失提炼需求说明的好处并降低长期的维护成本，请参考本章稍后9.4.3节附注栏的内容。

9.4.2 UI需求说明只检查UI功能

适用于：用户界面包含复杂逻辑时

→如果可执行的需求说明是以与用户界面元素的交互来描述的，那么请在需求说明中只叙述用户界面的功能。

在我接触过的团队中，以较低的技术层次来描述测试却又没有引起大量维护问题的团队仅有一个，就是BNP Paribasin银行的Sierra团队。他们有一组可执行需求说明，描述的是用户界面元素之间的交互。在其他故事中，这样的测试必定会惹出麻烦，但Sierra团队通过只说明用户界面的功能而不包括底层的领域业务逻辑避免了这种困境。例如，他们的测试检查表单的必填项以及使用JavaScript实现的功能。他们所有的业务逻辑需求说明都是在用户界面层之下自动化的。

毋庸置疑，提高抽象层次可以让此类测试更易于阅读和维护。但另一方面，这样做也会使得自动化层极度复杂化。由于这类测试的数量相对较少，因此创建并维护一个灵活的自动化层可能是得不偿失的，当用户界面发生变化时直接更改脚本可能会更加快捷。另外很重要的一点是，他们维护的是一个后台办公系统，它的布局不会像面向公众的网站，比如购物窗口那样频繁地改动。

9.4.3 避免录制的UI测试

许多传统的测试自动化工具提供“录制—回放”用户界面自动化的功能。虽然这听起来对初期的自动化很有吸引力，但是“录制—回放”对实例化需求说明来说是一个蹩脚的做法。这也是可执行需求说明的自动化区别于传统自动化回归测试的一个地方。

→如果可以，尽量避免录制用户界面自动化。录制的脚本除了几乎无法理解外，还难以维护。它降低了创建脚本的成本却会显著增加维护成本。

Pierre Veragen的团队曾经有7万行用户界面回归测试的录制脚本。为了与用户界面的重大变更保持一致，他们好几个人花了6个月的时间才重新录制完脚本。如此慢的反馈使得可执行需求说明的所有益处都完全失效。此外，“录制—回放”的自动化需要现成的用户界面，而实例化需求说明则是在编写软件之前就开始了。

有些团队一开始不理解传统回归测试与实例化需求说明的这个区别，他们尝试使用了“录制—回放”工具。Christian Hassa的故事就比较典型：

“这种测试太脆弱了并且维护开销非常大。**Selenium**测试是录制的，因此生成测试的时间还是太迟了。首先，我们试着在**sprint**末尾录制已经开发的功能。然后我们试着将所录制的脚本抽象成更具重用性且不那么脆弱的测试。但最终，还是要由测试人员自己决定测试的方法。我们要到很晚才了解测试人员是如何理解用户的期望的。同时，我们也落后于进度。由于我们必须不断维护所有这些内容，所以实际上情况越来越糟。**6**个月之后，我们使用的脚本已经无法维护了。

这种方法我们使用了几个月，并且还想方设法去做改善，但还是不管用，所以我们在项目末期将这些测试全部抛弃了。当时我们编写的测试结构不好，不像我们现在使用的组织方式，那时使用的是传统测试人员组织测试的方式：大量的前置条件，然后是一些断言，然后即将进行的事情又成为下一个测试的前置条件。**”**

用户界面自动化的**3**个层次

编写通过用户界面自动化的可执行需求说明时，请考虑使用以下**3**个层次来描述需求说明与自动化。

业务规则层。测试所展示的或所操作的是什么？例如：为购买**2**本或**2**本以上书籍的客户提供免费送货服务。

用户工作流层。用户如何通过**UI**使用某个功能，以更高的行为级别应该怎么描述？例如：将**2**本书放入购物车，输入地址信息，然后验证配送选项是否包含免费送货服务。

技术行为层。操作单个工作流的步骤需要哪些技术性步骤？例

如：打开店铺主页，使用**“testuser”**与**“testpassword”**登录，跳转到**“/book”**页面，点击**CSS**类是**“book”**的第一个图片，等待页面加载结束，点击购买链接等。

需求说明应该以业务规则层来描述。自动化层应该通过组舍技术行为层上编写的程序块来处理用户工作流层。这样的测试易于理解、编写高效，而且维护成本也相对较低。

关于**UI**测试**3**个层次的更多信息，请参考我的文章**“**编写**UI**测试时如何不搬起石头砸自己的脚**”**①。

注释：①http://gojko.net/2010/04/13/how-to-implement-ui-testing-without-shootingyourself-in-the-foot-2

9.4.4 在数据库中建立环境

→即使通过用户界面进行自动化是自动化可执行需求说明的唯一方式，许多团队发现通过直接在数据库里准备环境，可以显著提高测试的执行速度。

例如，当自动化一个描述编辑如何审核文章的需求说明时，我们可以调用数据库来预先创建文章。如果你使用前一小节附注栏中提到的3个层次，那么工作流层的某些部分可以通过用户界面来实现，而某些部分则可以使用领域API或调用数据来做优化。Ultimate软件公司的Global Talent Management团队就使用了这种方法，不过他们把工作拆分开来，让测试人员仍然可以有效地参与进去。Scott Berger是这么解释的：

“理想情况下，开发人员会利用数据库自动化层构建数据，他们会编写自动化代码快速打通主要通道，然后测试人员接手加入额外的用例对其进行扩展。**”**

通过尽早自动化这整个通道，开发人员可以利用自己的知识优化测试。有了第一个自动化的例子后，测试人员与分析师就可以很容易地扩展需求说明，在业务规则层加入更多的例子。

在数据库中建立测试的上下文环境，带来了第二大挑战，也正是我调研时采访的那些团队在自动化可执行需求说明中所面对的问题：数据管理。为了增强对系统的信心，或者由于他们的领域是数据驱动的，有些团队会把数据库包含在持续验证过程中。这为自动化带来了一组新的挑战。

9.5 管理测试数据

为了使可执行的需求说明具有专注和不言自明的特质，需求说明应

该包含所有重要的数据用以对功能进行举例说明，同时要省略其他信息。但对于一个使用数据库的系统，由于数据引用的完整性检查，我们通常还需要额外的数据才能完全自动化该系统的所有例子。

依赖于数据库数据的自动化测试还有另外一个问题，那就是某一个测试可能会更改另一个测试所需的数据，使得测试结果并不可靠。另一方面，为了获得快速反馈，我们又不能在每一个测试里删除和恢复整个数据库。

有效地管理测试数据是非常关键的，它可以增进我们对数据驱动的系统的信心，还可以让持续验证过程保持快速、可重复而且可靠。本节中，我将展示一些管理测试数据的好方法，我采访的一些团队就是用它们来管理其可执行需求说明的测试数据的。

9.5.1 避免使用预填充数据

适用于：说明非数据驱动的逻辑时

→重用现有数据可能会使得需求说明更加难以理解。

当自动化的可执行需求说明使用了数据库时，在数据库中的数据就成了自动化上下文的一部分。某些团队并没有在测试开始前将上下文信息自动放入到数据库中，而是重用符合其目的的已有数据。这可以让需求说明更容易自动化，却使得它们更难理解。阅读此类需求说明的人必须同时理解数据库中的数据。针对这一问题，Channing Walton这样建议道：

“通过预填充来建立一个拥有基准数据集的数据库总会带来很多痛苦。理解那些数据是什么、为什么需要那些数据以及它有什么用处是一件比较困难的事情。当测试失败时，也难以了解个中原因。由于数据是共享的，测试之间会互相影响。大家很快就会被搞得晕头转向。这是一种不成熟的优化。请编写与数据无关的测试。”

如果系统的设计方式并不需要设置大量的引用数据，那么自动化需求说明只需定义最少量的上下文信息。换个角度来看，实例化需求说明可以指导团队设计专注的、低耦合的组件，这也是面向对象设计原则里最重要的一条。不过，在遗留的数据驱动系统里很难做到这一点。

9.5.2 尝试使用预填充的引用数据

适用于：数据驱动的系统

由数据驱动的系统上下文很难准备齐全，而且也很容易出错。从需求说明要专注的角度来说，这么做也不见得是最好。Gaspar Nagy的团队曾经试着这么做，结果发现需求说明变得难以阅读和维护了：

“我们有一个验收测试需要在数据库准备数据才能执行。当我们编写了准备数据的描述后，发现它看起来像数据库表。虽然文本中没有

出现‘表’这个字眼，但确确实实是数据库表。这对开发人员来说很容易理解，但是对业务人员来说就难了。

例如，我们有一个‘国家’表。我们不想在测试自动化中硬编码任何‘国家’相关的逻辑，所以与此相关的所有测试都各自定义了‘国家’数据。结果我们都犯傻了，因为定义‘国家’的时候我们都只用了‘匈牙利’和‘法国’。其实我们可以在数据库里事先准备好全世界所有国家的数据，然后在测试里‘**Given**系统已经有一些缺省的国家数据’。事先准备默认的数据集本来会很有帮助的。”

Marco Milone也碰到过类似的问题，当时他在做新媒体产业的项目：

“起初，为了让测试运行，我们没有做到位。**Setup**和**Teardown**都在测试内部，很杂乱。之后，我们集中了数据准备并控制变更。测试只执行检测，无需准备数据。如此一来，测试变得更快、更易阅读并且更易管理。”

在数据驱动的系统中，重新创建所有的数据并不是一个好主意。同样，隐藏信息也会导致很多问题。对此，爱荷华州助学贷款公司的团队实施的策略是个不错的方案。他们只会事先准备那些不会变动的引用数据。Tim Andersen是这么描述这种方法的：

“我们在构建过程中清理并设置数据库，然后填充配置数据以及领域相关的测试数据，其余交易数据由每个测试自己负责创建并清理。”

→使用预填充的引用数据是个不错的策略，它可以让测试说明更加短小并更容易理解，与此同时它还可以加快反馈并简化自动化层。

如果你决定使用预填充的引用数据，请参考10.1.9节，了解一下如何让测试变得更加可靠。

9.5.3 从数据库获取原型

适用于：遗留的数据驱动系统

某些领域非常复杂，即使使用了预填充的引用数据，从头设置新的对象仍然是一项复杂而且容易出错的任务。如果你是在一个全新的项目里碰到这个问题，而且其领域模型也在你的控制之下，那么这可能是领域模型有问题的征兆（请参考第11章的11.4节）。

在遗留的数据驱动系统里可能无法修改模型。这种情况下，不需要完全从头开始创建新对象，只需在自动化层里克隆已有对象并修改相关的属性即可。Børge Lotre和Mikael Vik在挪威的奶牛记录系统里使用了这种方法。他们说：

“由于领域的复杂性，为测试准备尽可能完整的数据是个挑战。如果我们要测试“奶牛”的行为，却忘记了定义奶牛有**3**只小牛犊的测试用

例，那么在使用真实数据进行手动测试前，我们无法看到代码出错，也无法发现问题。所以我们创建了一个可以辨识真实‘奶牛’的数据生成器，它会从数据库中获取属性。然后这些属性会作为基础数据用在新的**Cucumber**测试里。这不仅会在我们需要重现错误时有所帮助，而且还会在我们开始编写新的需求时提供帮助。”

当Bekk的团队发现一个遗漏的测试用例时，他们会在真实的数据库里找出一个具有代表性的例子，并利用其属性让“数据生成器”为自动化的验收测试做准备。这样可以确保复杂对象拥有所有有关的数据并引用相关的对象，这会让验证检查更为确切。为了更快地从它们的可执行需求说明中获取反馈，数据生成器会获得一个对象的完整上下文，这使得测试可以运行在内存数据库中。

→在数据库中找一个具有代表性的例子，并用其属性来准备测试。

为测试做准备时，使用这种方法来快速创建对象而不是创建数据上下文（结合实际的数据库），还可以简化在可执行的需求说明中为相关实体做准备的工作。我们可以只说明那些对获取一个好的原型比较重要的属性，而不是说明一个对象的所有属性。这会让需求说明更容易理解。

自动化验证而不修改需求说明，与传统的测试自动化具有概念上的区别，这就是为何有如此多的团队在刚开始实施实例化需求说明时颇费周章的原因。我们对需求说明进行自动化是为了获得快速反馈，但我们的主要目的应该是创建易于访问、简单易懂的可执行需求说明，而不仅仅是为了自动化验证过程。一旦需求说明可以执行，我们就可以频繁地对其进行验证，建立起一个活文档系统。我们将在接下来的两章中涉及这些概念。

9.6 铭记

自动化提炼好的需求说明时必须尽量减少改动。

自动化层应该定义如何进行测试，而需求说明则应该定义要测试什么内容。

使用自动化层执行业务语言与用户界面、API和数据库之间的转换。为需求说明创建更高级别的可重用组件。

尽可能在用户界面之下进行自动化。

除非万不得已，否则不要太依赖于现存数据。

第10章 频繁验证

“偏离车道时，响亮的振动声立刻就会引起你的注意。”

——David Haldane[①]

注释：①http://articles.latimes.com/1997-03-07/local/me-35781_1_botts-dots

在20世纪50年代，美国加州的交通部深受机动车车道线问题的困扰。车道线会褪色，每个季节都需要重新喷涂。不仅费用高昂，还会扰乱交通，而且对喷涂的工人来说还很危险。

Elbert Dysart Botts博士致力于解决这个问题，他做了很多试验，试图寻找更加反光的涂料，但最终都证明那是死路一条。跳出条条框框，放开思路后，他发明了突起的车道标志，称作Botts点。无论什么天气，白天黑夜都能看得见Botts点。它们不会像喷涂的车道线一样容易褪色。当司机越过指定的车道时，除了视觉，司机还可以感受到Botts点产生的振动以及隆隆的振动声。已经证明这种反馈是高速公路上最重要的安全特性之一，当漫不经心的司机偏离车道时可以警告他们即将发生危险。

Botts点作为最早的12个极限编程实践之一引入到软件开发中，称作持续集成(CI)。当疏忽大意的软件团队开始偏离产品必须时时可构建和可交付的原则时，持续集成就会发出警示。专用的持续集成系统会频繁地构建产品并运行测试以确保系统不仅仅在开发人员的机器上运行正常。这一实践通过快速地找到潜在问题让我们保持在正确的道路上，并在必要时只要采取小而廉价的措施就可以纠正。持续集成可以确保产品在被正确地构建起来后，能始终保持正确。

同样的原则也可以应用到构建正确的产品上。一旦构建了正确的产品，我们就要确保它一直正确。当它偏离设计的方向时，只要我们能尽早地知道，就可以更容易更廉价地解决问题，不让问题累积起来。我们可以频繁地验证可执行的需求说明。持续构建的服务器[②]可以频繁地检查所有的需求说明，并保证系统始终符合需求。

注释：②当有人改变版本控制系统中的任何代码时，进行自动构建、打包、运行测试的软件。如果你从没有听说过，请用Google搜索Cruisecontrol、HudSon或TeamCity。

持续集成是一个有完备文档记录的软件实践，已有许多作者对它作了详尽的描述。我不想笼统地重复如何搭建持续构建与集成的系统，但是有些关于可执行需求说明频繁验证的难点对于本书的主题十分重要。

许多团队使用实例化需求说明来扩展现有系统，他们发现可执行的

需求说明必须运行在真实1的数据库上，有真实的数据、外部服务或完整部署的网站。功能验收测试通过许多组件来检查功能，如果系统事先不具有可测试性，那么此类检查往往需要集成部署整个系统才可进行。除了一般持续集成用到的技术性的（单元）测试，上述情况还会带来其他3组关于频繁验证的问题：

依赖环境而导致不稳定。单元测试一般独立于测试环境，但是可执行需求说明可能强烈依赖于它们运行的生态系统。即使代码正确，环境问题也会导致测试失败。为了对验收测试结果更有信心，我们必须解决或减少这些环境问题，使测试执行更为可靠。

反馈较慢。遗留项目的功能验收测试运行速度通常比单元测试慢很多。如果单元测试需要运行几分钟，我就认为它是比较慢的了。将近10分钟的单元测试绝对是太慢了，我会很认真地研究如何提高速度。另一方面，我见过许多验收测试需要长达数小时的运行时间，除非牺牲测试的可信度，否则很难进行优化。整体反馈是如此之慢，我们需要想出一种解决方案，使得系统某些指定的部分可以按要求提供快速反馈。

对失败测试的管理。大量粗粒度的、依赖于许多活动部件的功能测试，要求一些团队（特别是那些刚开始实施实例化需求说明的团队）去管理失败的测试而不是直接进行修复。

本章我将解释我调研过的这些团队是如何处理这3个问题的。

10.1 提高稳定性

不稳定的验证过程会削弱团队对产品和实例化需求说明过程的信心。调查那些间歇性的又不是由真正问题导致的失败会浪费大量的时间。如果这种事情经常发生，开发人员就有了根本不去查看验证问题的借口。这会让真正的问题逃过检测，使整个持续验证变得没意义。

遗留项目基本无法很容易地做自动化功能测试，所以可执行的需求说明可能需要通过不可靠的用户界面来自动化，或者要在异步流程带来的不确定性中苦苦挣扎。当我们需要说服开发人员参与到过程中去，而开发人员只将其看作是功能测试的改进版（换句话说，那不是他们的问题）时，问题变得尤其棘手。

Clare McLennan和她的团队就碰到了这样的问题。“因为测试本身不稳定，所以开发人员不关心这些测试。然而我们就是需要开发人员的知识来使测试变得稳定。”对她的团队来说这是个鸡生蛋蛋生鸡的问题。为了让开发人员参与进来，她必须让开发人员看到可执行需求说明的价值。但是，只有稳定可靠的可执行需求说明才能展现出它的价值。而需求说明要想稳定可靠，则必须要开发人员修改系统设计，使系统易于自

动化测试时才能实现。

为了获得实例化需求说明的长远收益，许多团队不得不投入大量的精力让验证过程变得可靠起来。本节我将阐述关于这方面的一些很好的想法。

10.1.1 找出最烦人的问题并将其解决掉，然后不停地重复

适用于：系统对自动化测试支持得不够好时

要想在系统上更加可靠地运行自动化测试，最重要的一点是要明白这样的改变不是一蹴而就的。要改变一个遗留系统不容易，否则它也不会叫做“遗留”系统了。当某种东西在多年的构建过程中都没有丝毫的“可测试性”设计时，想要瞬间变得清晰可测试是不可能的。

快速地引入太多重大修改会使系统变得不稳定，尤其是在功能测试覆盖率不高的时候。它会严重打乱开发流程。

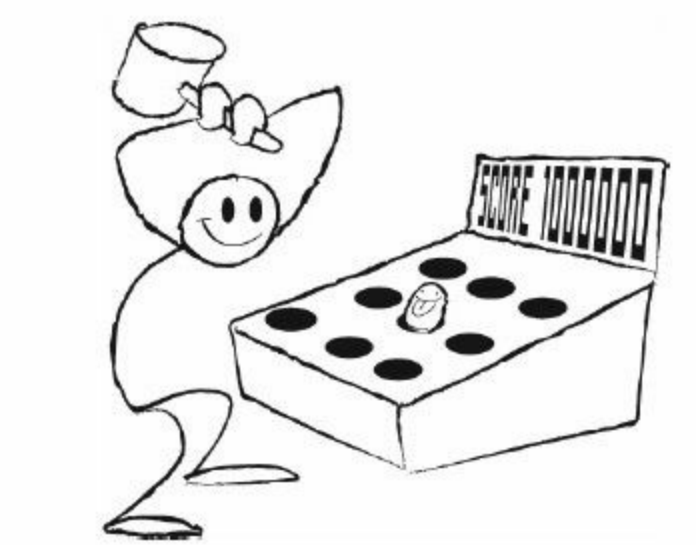
→不要想一下子就解决问题，更有效的策略是一个迭代一个迭代地做些小的改变。

比如，McLennan的团队意识到缓慢的测试数据处理会导致测试超时。他们的数据库管理员改善了数据库性能，这让他们发现有些测试在测试数据更新前就开始了，所以系统一直在用老数据。他们引入了消息机制告诉测试何时数据库里准备好了最新的数据，之后他们就可以可靠地开始测试，避免假漏报率。当不可预测性的源头解决之后，他们发现HTTP cookie过期导致了一些问题。于是他们引入了业务时间的概念，这样他们就可以修改系统时间。（见本章稍后的10.2.1节。）

作为提高自动化测试稳定性的策略，McLennan提出了增量方法：

“找到最烦人的问题并将其解决掉，然后别的问题又会变成最烦人的问题，解决之后，目标再次转向另一个最烦人的问题。如此不断重复，最终的系统将会非常稳定，非常有用。**”**

一个迭代一个迭代地提高稳定性是构建可靠验证过程的好方法，同时又不会过多地妨碍交付流程。这种方法还可以让我们在提高系统可测性的过程中不断学习并适应。



10.1.2 用CI测试历史找到不稳定的测试

适用于：在遗留系统中引入自动化测试时

对一个不具有自动化测试设计的遗留系统来说，由于存在太多导致不稳定的地方，想要做增量式清理，往往还是无从下手。有个不错的方法是查看测试的执行历史。当今大多数持续构建系统都可以跟踪一段时间的测试结果。

→只要可执行需求说明在持续构建(CI)系统中运行，我们就可以从测试的运行历史中看到哪些测试最不稳定。

多年以来我完全疏忽了这个功能，因为我参与的大多是事先考虑了可测试性的新项目，或者是那些只需少量的改动就很稳定的系统。在这些项目中，跟踪测试执行历史没什么用：测试几乎一直都是通过的，即使失败也很快就修复了。我第一次尝试改进自动化测试的系统，是一个偶尔有超时问题、网络问题、数据库问题以及不一致问题的系统，查看测试历史能帮助我集中精力提高稳定性。这个功能告诉了我哪些测试最经常失败，因而必须最先修复。

10.1.3 搭建专用的持续验证环境

→如果你的应用程序需要部署，并且要运行功能测试，那么保证可重现性的第一步就是确保专用的部署环境。

持续验证必须以可重现的方式进行工作，这样才可靠。在一些大型的组织中，获取一批新机器要比雇用有名的投毒杀人犯担任公司自助餐厅的大厨还要难，但是争取更好的环境依然是值得的。许多团队将一个环境用作不同的用途，比如为商业用户演示功能、手工测试以及持续验证。这往往会导致数据的一致性问题。

没有专用的环境，很难知道测试失败是由于存在缺陷、由于有人对测试环境做了改变，还是由于系统不稳定所导致的。专用的环境可以排除计划外的改变，并可以降低环境不稳定的风险。

10.1.4 使用全自动部署

一旦我们有了专用环境，我们要确保软件以可重复的方式部署。不可靠的部署是测试结果不稳定的第二个最常见的原因。对于许多遗留系统，部署需要整晚的时间，需要多名人员、大量的咖啡，最好还要有个魔法棒。如果我们每年只需部署一次，还是可以接受的。然而，当我们需要每两周部署一次时，这就是一件非常头疼的事情了。为了持续验证，我们可能需要一天部署几次，需要魔法相助的手动部署是完全难以让人接受的。

没有全自动部署来可靠地升级系统，我们就会频繁地碰到这样的情况：许多测试突然开始失败，有人得花上数小时寻找罪魁祸首，结果只会听到房间后面有人说“但是这在我机器上是好的”。

→全自动部署可以确保升级只有唯一的标准流程，同时可以确保所有的开发人员拥有和测试环境一样的系统部署。

这排除了可执行需求说明对特定环境的依赖，可以大幅提高持续验证的可靠性。同时也更容易排除故障，因为开发人员可以用任何环境来重现问题。

部署必须完全自动化才行。完全不需要或不允许手动干预。（注意我是说可以按需执行的全自动部署，没必要自动触发部署。）如果安装程序需要处理管理员控制台或者半自动化的手工脚本，那么它就不算完全自动化。特别值得一提的是，全自动部署还包括自动化的数据库部署。

我见过许多团队声称自己有自动化部署，结果发现需要有人在之后手动运行数据库脚本。

全自动部署还可以带来其它好处，比如可以更容易地升级生产环境。长远来看，这将为你节省大量的时间。不管是否使用实例化需求说明，频繁部署都是一个非常好的实践。

10.1.5 为外部系统创建较简单的测试替代品

适用于：有外部参考数据源时

许多团队处理业务流中的外部参考数据源或外部系统时都碰到了问题（我所说的“外部”是指超出团队的范围，不一定是属于不同组织的）。大型企业有着复杂的系统网络，团队负责的工作流可能只是系统的一部分，而测试系统则需要和其他团队的测试系统相沟通。问题是其他团队要做他们自己的工作和测试，所以他们的测试服务器不可能始终

保持可用、可靠并且正确。

→创建一个单独的假数据源来模拟与真实系统的交互。

Rob Park的团队在一家大型的美国保险公司工作，他们构建的系统需要从外部的自动策略服务器查询相关策略数据。如果自动策略服务器当机了，他们所有的可执行需求说明就会失败。对于功能测试，他们使用了一个外部服务的替代版本。这个版本更加简单，可以从本地磁盘上的文件读取数据。

这让Park的团队即使在自动策略服务器下线的时候也可以测试他们的系统。创建一个单独的引用数据源还能使团队完全控制系统引用的数据。而且真实系统无法提供已经过期的策略，所以那些依赖于过期策略的测试就会失败。

引用数据源的简化版本从配置文件里提供了所有东西，这避免了时效性问题。他们把数据存放在一个XML文件中，并将其签入到版本控制系统中，这样他们就可以很容易地跟踪改变，并能够把相应版本的测试数据和代码打包在一起。而外部系统是不可能这样做的。读取文件的本地服务也比外部系统快，使得整个反馈更为迅速。

使用测试替代品的风险是真实系统会随着时间演化，替代品会无法反应真实的功能。为避免这个问题，要确保执行阶段性检查，看看替代品是否像原始系统一样工作。这很重要，尤其是当替代品代表着你无法控制的第三方系统的时候。

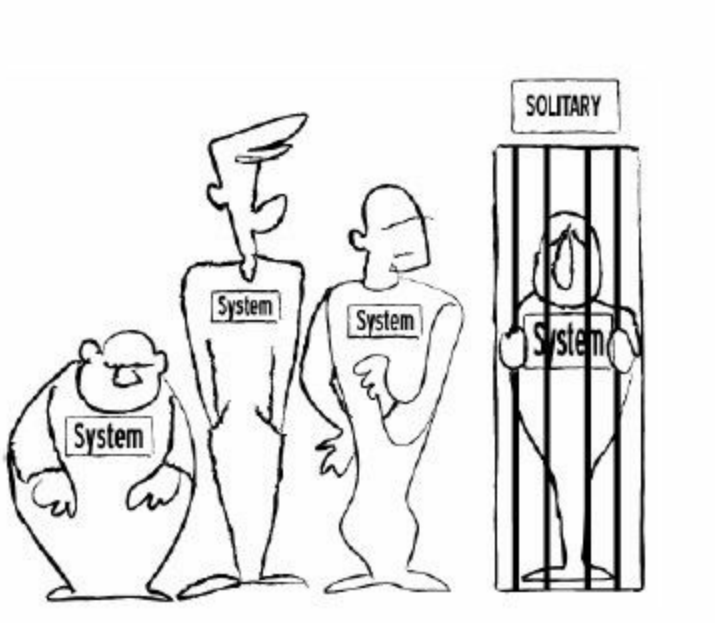
10.1.6 选择性地隔离外部系统

适用于：有外部系统参与其中时

完全隔离一个系统的做法并不见得永远适用。当系统参与到一个更大的工作流中，外部系统提供的不再仅仅是引用数据的时候，测试替代品就必须开始实现外部系统的部分真实功能。这会带来巨大的开发成本和更多的维护问题。

在Beazley，Ian Cooper的团队采用一个有趣而实用的方法解决了这个问题。他们根据每个可执行需求说明的目标，选择性地禁止访问某些服务。这使他们的测试速度有显著的提高，但是每个测试仍然会涉及一些必须的真实外部系统。这个方案并没有完全帮他们从外部影响中解救出来，但是这使故障排除简单多了。如果测试失败，可以很清楚地知道哪些外部依赖影响了它。

→选择性地隔离一些外部服务可以让测试更快，故障排除更容易。



10.1.7 尝试多级验证

适用于：大型、多级组织

在遗留系统中，运行所有的可执行需求说明所花的时间往往要比两次提交代码修改的间隔时2间还要长。因此，将某个问题关联到导致该问题出现的某次修改可能是比较困难的。

有多个团队的时候，特别是当他们分布在多个不同地方的时候，这会导致问题不断积累。3如果一个团队破坏了数据库，那么其他的团队将无法验证他们的修改，直到问题被修复。发现存在问题、确定是何问题、修复它并重新运行测试以确认修复成功，这些可能要花上数小时时间。失败的构建总是别人的问题，很快持续验证测试将总是失败。到那时，我们可能就会停止运行测试。

→要使用多级验证。每个团队都应该有隔离的持续验证环境，任何变更都应该先在那里进行测试。

首先要整合同一个团队的修改。如果测试通过，则把修改推送到持续验证的中心环境中，在那里与其他团队的变更进行集成。

大多数情况下，这种方法可以防止一个团队的问题影响到其他团队。即使中心环境被破坏了，有人正在进行修复，各个团队仍然可以使用他们自己的环境来验证他们所做的变更。

根据运行所有可执行需求说明所花的时间，我们可以选择在这两种环境中都运行所有的测试，或者只在其中一个环境运行一部分典型的测试作为一次快速的冒烟测试。比如Ultimate软件公司的Global Talent Management团队，他们的测试大多只在本地团队环境中运行。较慢的

测试不要为获得快速的反馈而在中心环境中运行。

10.1.8 在事务中执行测试

适用于：可执行需求说明需要修改引用数据时

→数据库事务可以隔离外界的影响。

事务可以防止我们的进程影响到其他正在运行的进程，并提高测试的可重现性。

如果我们在测试中创建一个用户，那么下次我们运行这个测试的时候它可能会失败，因为数据库中存在用户名的唯一性约束。如果我们在事务中运行测试，并在测试最后进行回滚，那么用户数据不会得到保存，两次测试执行会是独立的。

在许多情况下这是一个不错的实践，但是在某些事务相关的上下文中（比如如果数据库的约束检查延迟到事务提交之后，或者在嵌套的自治事务中），这可能是行不通的。这些高级的事务话题不在本书的讨论范围内。

任何时候都要在需求说明外部控制事务。数据库事务控制是一个跨领域问题，最好在自动化层实现，而不是在可执行需求说明中描述。

10.1.9 对引用数据做快速检查

适用于：数据驱动的系统

在数据驱动的系统中，可执行需求说明广泛依赖于引用数据。对引用数据的变更（如工作流配置）即使在功能正确的情况下也有可能破坏测试，这样的问题很难解决。

→专门设置一组完全独立的测试来验证引用数据是否与我们期望的一致。

这些测试可以在可执行需求说明运行之前快速地运行。如果这些测试失败，就没有必要运行其他的了。这些测试还会指出引用数据的问题，让我们可以快速地进行修复。

10.1.10 等待事件，而非等待固定时长

异步过程似乎是可执行需求说明中问题最多的一个领域。甚至当一个过程的一部分在后台执行、运行在不同的机器上，或者需要数小时的延迟时间时，商业用户依然会把整个过程看成是单个事件序列。在Songkick，这是他们成功实施实例化需求说明的一个最大的挑战。Phil Cowans说：

“异步过程真的令我们很头疼。基于性能原因，我们有很多后台处理过程是异步的，而由于测试是即时运行的，我们碰到了很多问题。测试进行到了下一步，而后台处理还未执行。”

要可靠地验证异步过程，需要在自动化层（通常是在生产系统中）进行一些计划和细心的设计。对异步系统的测试，我们经常会见到的错误是花一定时间等待某件事情发生。症状之一是存在“等待10秒”这样的测试步骤。出于几方面的原因，这样做很糟糕。

即使程序功能都正常，当持续验证的环境负荷较大的时候，这些测试仍可能会失败。在不同的环境中运行这些测试可能需要更多的时间，因此它们开始依赖于特定的部署。当持续验证的环境比开发人员的机器强劲得多的时候，在同一个超时配置(Timeout Configuration)下，开发人员的机器就无法去验证变更。许多团队会为测试设置较高的超时时间，以使测试对环境变化的适应力更强。于是这些测试产生了不必要的反馈延迟。

例如，如果一个测试无条件地花一分钟等待某个过程结束，但是那个过程只用10秒就完成了，我们就产生了50秒钟不必要的反馈延迟。较短的延迟对单个测试可能不成问题，但是它们在测试包中会累积。进行20个这样的测试，整个测试的反馈就会延迟15分钟以上，这个影响就很大了。

→去等待一个事件的发生，而不要等待一段时间的流逝。这会使测试可靠得多，而且不会造成不必要的反馈延迟。

可能的话，让这些测试去阻塞消息队列，或频繁轮询数据库（或后台服务），以检查处理过程是否已经完成。

10.1.11 将异步处理变成可选

适用于：新项目

从零开始构建一个系统的时候，我们可以将其设计成容易进行测试。根据配置，系统可以在队列中添加一个消息以便让后台去处理事务，或者也可以直接执行它。我们可以配置持续验证环境来同步执行所有的过程。

→有个提高测试稳定性的好方法，就是将异步处理作为一个可选项。

这个方法可以让可执行需求说明运行得更快更可靠。但是这也意味着功能测试不会端到端地检查系统。如果你在功能测试中关闭了异步处理，要记得编写额外的技术测试来验证异步处理是否正常工作。虽然听起来这可能使工作量加倍了，但实际上却并非如此。技术测试可以比较短小，而且可以只专注于技术协调，无需验证业务功能（业务功能将单独在功能测试中检查）。

还有一些很好的自动化验证异步处理的技术方案，可以参考Growing Object Oriented Software，Guided by Tests一书。

10.1.12 不要用可执行需求说明做端到端的验证

适用于：改造项目

许多团队，特别是在现存的遗留系统上工作的团队，既用可执行需求说明做功能测试，又用它做端到端的集成测试。这使他们更加自信软件整体上工作正常，但是反馈却慢了很多，并且显著增加了测试的维护成本。

这么做的问题在于它同时测试了太多东西。这样的端到端测试会检查过程的业务逻辑、与技术底层的集成情况以及所有组件之间能否互相通讯。活动部件太多意味着其中任意部分的改变将会导致测试失败。这还意味着即使我们只想测试整个过程的一小部分，我们也不得不端到端地运行整个过程来验证每种情况。

→不要在一个大的可执行需求说明中同时测试太多东西。

大多数异步过程由几个步骤组成，每个步骤都有一些业务逻辑，这些逻辑需要用可执行需求说明来定义。它们大多还包括一些单纯的技术任务，比如分发到一个队列，或者保存到数据库。与其使用一个较大的可执行需求说明花很长时间去检查所有事情，倒不如在实施这个过程的时候考虑一下以下想法。

明确分离业务逻辑和底层代码（例如，加入到一个队列、写入数据库）。

如本章之前所说的，分别说明并测试每个步骤中的业务逻辑，如果可能的话隔离底层代码。这会让编写和运行测试简单得多。测试可以同步运行，它们不需要和真实的数据库或真实的队列进行交互。这些测试能让你确信是否正确地实现了业务逻辑。

为底层代码、队列或数据库实现的仓库编写技术集成测试。这些测试可以很简单，因为它们不需要验证复杂的业务逻辑。它们可以让你确信是否正确地使用了底层设施。

编写一个端到端的集成测试去验证所有组件都能正确地进行交互。可以执行一个简单的业务场景，它必须涉及所有的组件，比如阻塞一个队列等待响应，或者轮询数据库检查结果。这可以让你确信配置是否正常。如果你使用高层次的例子，就像5.2.3节中建议的，那么它们会是不错的候选测试。

从底层业务逻辑的测试到端到端的测试，一般我期望看到的是测试数目逐步减少，而执行单个测试的时间则显著增加。通过把复杂的业务逻辑从底层中剥离出来，我们就能提高可靠性。

10.2 获得更快的反馈

大多数团队都会发现，每次系统改变后都执行所有的可执行需求说明不太现实。如果测试包含大量的检查（特别是这些检查只能对网站、数据库或外部服务运行时），那么从完整的测试执行中获得反馈就太慢了。为了更有效地支持开发并辅助变更，大多数团队都对他们的持续验证系统进行了改动，以便提供分阶段的反馈，这样他们就能更快地得到最重要的信息。这里有一些我采访过的团队使用的策略，它们能有效缩短反馈时间。

10.2.1 引入业务时间

适用于：有时间约束时

时间约束是自动化功能测试反馈慢的一个常见原因。通常，每天结束后的任务(job)可能会放到半夜运行，任何依赖于它们的测试都会产生较慢的反馈，因为我们必须平均等上12个小时才能看到结果。缓存头(Cache headers)可能会影响一个文档是否从后台获取，要正确测试这个功能我们可能需要等上几天才能得到结果。

这个问题有个很好的解决方案，那就是在系统中引入业务时间的概念，它是一个可配置的时间，可以代替系统时间。

→当系统需要获取当前时间或日期的时候应该使用这个业务时钟。这让我们可以在测试进行的时候很容易穿梭于各个时间之间。这会增加一点系统的复杂度，但是它可以让我们的测试快很多。

实现业务时间有个快速丑陋的方式是在测试环境自动修改时间，这并不需要改变设计。注意有些应用会缓存时间，因此当系统时间更新后它们可能需要重启。如果其他地方也在使用这个时间，比如，在测试执行报告中，那么修改系统时间可能会让测试结果更难理解。一个更好的方案是在生产软件中增加业务时间的功能。

引入业务时间还可以解决过期数据的问题。举例来说，使用即将过期的合同进行测试在6个月之内可能一直正常工作，然后当合同过期时就突然开始失败了。如果系统支持修改业务时间，我们就可以保证这些合同永远不会过期，以此减少维护成本。

引入业务时间的潜在风险是当系统需要同我们不能影响的外部系统进行通讯时，会引发时间的同步问题。要解决这个问题，请使用本章之前讲到的测试替代品或采用隔离的方法。

10.2.2 将较长的测试分割成较小的模块

在构建可执行需求说明数月乃至数年之后，许多团队的测试包需要运行数小时。因为有数以百计的测试要执行，所以大家会想当然地认为反馈慢是很正常的。他们不会注意到有些测试比以往多花了20钟。这些问题会快速累积，而反馈就需要更长时间了。

→相比需要运行**6**个小时的一组大的可执行需求说明，我更希望运行**12**组较小的测试，每个花费不到**30**分钟的运行时间。通常，我会把这些测试按功能分解。

比方说，如果我要解决一个会计子系统的问题，我只需快速地重新运行会计相关的测试就可以检查问题是不是已经解决。我不必花费6个小时去等待所有其他测试的完成。

如果某一组测试的执行时间突然增加了10分钟，我通过查看特定测试组的测试历史，很容易就能发现这一点。与半个小时增加10分钟相比，6个小时增加10分钟没那么容易察觉。这可以让反馈延时保持在可控的范围内，因为我会开始想办法优化特定的测试组。最近，通过将一个很大的测试组分解成几个较小的测试组，我帮助一个客户意识到，只有一个功能领域会导致很长的延时。我们把将近一个小时的时间降低到了仅仅9分钟。这就是“分而治之”的妙处！

10.2.3 避免使用内存数据库做测试

适用于：数据驱动的系统

为了加快反馈，有些团队做数据驱动系统的时候会把持续验证测试运行在内存数据库上而不是真实的数据库上。这样系统仍然可以执行SQL代码，但这会让所有的SQL调用运行得快很多。这也可以更好地隔离测试的运行，因为每个测试可以用各自的内存数据库来运行。但实际上许多团队发现，长远来看像这样运行测试成本会高于它所带来的收益。

包括数据库部分的可执行需求说明很像功能验收测试，同时又很像

端到端的集成测试。编写这种测试的人一般还需要检查SQL语句的执行。只要真实的生产数据库和内存数据库实行之间存在着细微的SQL语言差别，就可能导致错误的测试结果。这种方法往往还需要维护两套SQL文件，并且两个地方都需要进行数据变更的管理。

→如果你使用内存数据库，端到端的集成测试就会去验证系统是否能在内存数据库（而不是你在生产环境中用的真实数据库）上工作正常。

我之前提到过一些更好的解决方案可以解决这个问题。在事务中运行测试可以提供更好的隔离性。混合端到端的集成测试和功能验收测试可能不是一个最好的主意（如本章之前所提到的），但是如果你真要这么做，那么请使用真实的数据库，并想办法提高它的速度。

爱荷华州助学贷款公司的团队使用了内存数据库做测试，但是后来他们又放弃了。Tim Andersen说：

“我们使用**SQL Server**作为数据库，并使用**Hypersonic**代替它运行在内存中。这为我们**45**分钟一次的构建过程节约了**2**分钟。一旦我们在数据库中添加了索引，**SQL Server**确实更快了。**Hypersonic**需要更多的维护，但它又没有加快多少构建时间。**”**

如果测试在真实数据上运行得很慢，那就意味着系统在生产环境中运行可能也会很慢，因此提高测试数据库的性能从长远计划来说也是有意义的。注意这条建议只适用于数据驱动的系统，这样的系统上经常需要执行特定的数据库代码。使用内存数据库来加速不确定数据库的检查可能会是非常好的解决方案。

10.2.4 把快速的和缓慢的测试分开

适用于：一小部分测试占用了整个测试的大部分执行时间时

→如果一小部分测试的运行时间占据了整个测试的大部分时间，那么只去频繁地运行那些较快的测试也许是个不错的主意。

许多团队将他们的可执行需求说明根据执行速度分成2～3组。比方说在RainStor，为了检查系统的性能，运行有些测试需要非常大的数据量。他们有一组功能测试每次构建都会运行，所需时间不到1个小时。到晚上，他们还会使用从客户那里得来的真实数据来运行客户场景。他们每个周末运行一次耗时很长的测试。虽然这种方法不能每次都提供完整的验证，但这能显著降低变更引入问题的风险，同时还能提供相对较快的反馈。

10.2.5 保持夜间测试的稳定

适用于：缓慢的测试只在夜间执行时

延迟执行缓慢的测试有个比较严重的问题，那就是无法很快地发现

和修复这些测试中存在的问题。如果夜间执行的测试包失败了，我们到早上发现后试着去修复问题，然后要到下一个早上才能得到结果。这么慢的反馈会使夜晚构建经常失败，而这会掩盖白天引入的其他问题。

→只把不容易失败的测试放到夜晚执行。

为了保持夜间测试的稳定性，可以把失败的测试加入到另一个测试包中（参考本章稍后面的10.3.1节）。

另外一个方法是只把那些已经可以稳定通过的测试加入到夜间测试包中。测试运行的历史统计数据（本章前面讨论过）可以帮助我们判断测试是否已经足够好，可以放入到夜间测试包中。

如果这些测试太慢而不能持续地执行，有个可能的办法是将它们周期性地按需执行，直到它们变得足够稳定可以延期执行。RainStor的Adam Knight使用了这样的策略：

“手动执行测试，直到它们足够稳定，然后再放入到夜间测试包中执行。把测试分开可以给我们带来许多方面的好处。如果测试失败，我们就修复它。严重的测试失败会成为我们的最高优先级。”

只在测试不太可能失败的情况下，才把需求说明加入到夜间测试包中，通过这种方法，RainStor的团队降低了慢反馈测试失败的风险，这时候失败是由于功能还处于开发过程中。这显著地降低了夜间测试包的维护成本。它们还是会捕捉到没有想到或无法预计的改变，虽然这种情况很少见。考虑到这一点，为了降低维护成本，这些功能的慢速反馈是一种很好的折中做法。

10.2.6 为当前迭代创建一个测试包

将运行时间较长的测试分割成较小的测试包有一个常见的特殊情

况，那就是为当前的迭代创建测试包。这个测试包包括当前开发阶段会影响的可执行需求说明。

→当前迭代的变更，会影响系统中最不稳定和最重要的部分，清晰地划分出当前迭代测试包可以让我们快速地获得有关这部分反馈。

如果我们把当前迭代的测试包从其他测试中拆分出来，那么即使是针对于那些已计划而尚未实现的功能而编写的测试，我们也可以放心地将它们加入进来。运行当前迭代的所有测试可以让我们更容易地跟踪开发进度，准确地了解我们何时可以完成。大部分时候，当前迭代的测试包可能会整体失败，但是这并不会影响主要的回归验证。

如果我们需要更为频繁地验证这些需求说明，那么可以变通地使用这种方法，那就是为当前发布创建一个测试包。注意许多自动化工具允许我们创建并行的层次结构，如此一来同样的需求说明可以同时隶属于多个不同的测试包。

10.2.7 并行运行测试

适用于：拥有多套测试环境时

→如果你在公司有权建立多套测试环境，那么只要你把大的测试包分割成多个小的集舍，你就可以试着并行运行它们。这会让你获取到最快的反馈。

如果有些测试必须隔离运行，不能并行执行，那么可以把它们也分割到一个单独的集合中。在LMAX，jodie Parker是这样组织持续集成和验证的：

“提交构建**(Commit Build)**在**3**分钟内就可运行完所有的单元测试并进行状态分析。如果通过了，顺序执行的环境会执行需要独立运行的或不能并行运行的测试。然后**23**个虚拟机会并行运行验收测试。在这之后，性能测试就开始执行。它们（可执行需求说明）一般运行**8**～**20**分钟。

最后，如果一定数量的测试通过了，我们会部署一个**QA**环境运行冒烟测试和探索测试，并给开发人员提供反馈。如果提交构建失败了，大家就得停下来（一项完全的禁令）将其修复。**”**

如果你们对安全不那么敏感，或者工作中没有监管约束阻止你们把代码部署到组织外，那么新兴的云计算服务可以帮助你运行并行测试。远程部署代码会花一些时间，但是这可以让你同时在大量的机器上运行测试。在Songkick，他们使用亚马逊的EC2云来运行验收测试。Phil Cowans说这帮助他们大幅降低了构建时间：

“在单个机器上运行完整测试需要**3**个小时，但是我们采用并行执行的方法。我们刚刚学习了如何在**EC2**上执行测试，把时间缩短到**20**

分钟。”

有些持续构建系统，比如TeamCity[①]，已经提供了在EC2上执行测试的标准功能。这让我们可以更加简单地使用云计算进行持续验证。还有些新兴服务通过云服务提供自动化，比如Sauce Labs就值得一试。

注释：①http://www.jetbrains.com/teamcity

10.2.8 禁用风险较低的测试

适用于：测试反馈非常慢时

uSwitch的团队对运行时间较长、反馈较慢的测试包有其独特的处理方式。一旦相关功能实现后，他们就会禁用与之对应的风险较低的测试。Damon Morgan说：

“有时候，你编写的验收测试（用来驱动开发非常好用）一旦在功能开发完成后就不太重要了。有些东西并不赚钱（比如，发送延迟的邮件），这不是网站很核心的部分，只是附加的一些功能……它们（可执行需求说明）可以很好地帮助我们驱动开发，但是之后持续地将它们作为回归测试来运行对我们并没有什么太大的用处。维护它们比直接扔掉繁琐多了。在我们的代码控制系统中的确有一些测试是不会运行的。当我们需要扩展功能的时候，我们依然可以去修改已有的测试。”

对我来说，这是一个很有争议的做法。我认为如果一个测试值得运行，那就值得一直运行下去[①]。

注释：①其实，我是从David Evans那儿得到的这个想法。所以如果你想引用，请把David作为引用出处。

在uSwitch，大多数测试是通过网页用户界面运行的，所以它们需要很长的运行时间，维护成本也很高，正是这个原因驱使它们采用了这种方法。对于一个没必要总是运行端到端测试的系统，我更喜欢尝试不同的方法去进行测试自动化，这样它们的成本就不会很高（请参考9.3.6节）。

uSwitch可以选择禁用测试的一个原因是他们有单独的系统在监控着他们生产网站的用户体3验，该系统能侦测出网站用户是否看到错误或者某个特定功能的使用率是否突然下降。

讲完了更快的反馈和更可靠的验证结果后，现在该处理更具争议性的问题了。许多团队开始4意识到，随着活文档系统的增长，他们偶尔还是要忍受失败的测试。在下一小节，我会介绍一些好的方法来处理那些不大容易马上修复的失败测试。

10.3 管理失败的测试

在Bridging the Communication Gap一书中，我说过失败的测试不应该直接禁掉，而应该马上修复。在为本书做完调研后，我略微改变了我的看法。有些团队在持续验证中拥有成百上千个检查，这些检查会验证花了数年才构建出来的功能。从Pierre Veragen那里得知，Weyerhaeuser的计算引擎系统（第1章提到过）一直由30000多个检查持续验证着。

频繁验证这么多的需求说明会发现许多问题。另一方面，运行这么多的检查往往意味着反馈很慢，所以问题不会马上被发现或解决。有些问题可能还需要商业用户的澄清，或者它们的优先级相比当前迭代将要上线的功能可能会来得低，所以我们没必要在发现这些问题后马上就全部修复它们。

这意味着持续验证中有些测试会失败，并且会失败一段时间。当测试包失败的时候，人们往往不会去寻找它们可能导致的其他问题，所以将这些测试置之不理也不是个好办法。以下是一些对系统的功能性回归测试进行管理的窍门。

10.3.1 创建已知失败了的回归测试包

类似于为当前迭代创建单独的可执行需求说明，许多团队专门创建了预期会失败的测试的测试包。

→当你发现回归测试失败了，并且你不决定马上去进行修复，那么请把相关测试加入到单独的集舍中，这样即使测试失败了也不会影

响主要的验证集舍。

把失败的测试单独放在一起可以让我们跟踪此类问题的数量，防止对暂时失败的“故意放松”规则成为导致问题堆积的“免费出狱”卡。

单独的集合还能让我们阶段性地运行所有失败的测试。即使测试失败，它们还是值得运行的，因为这可以检查是否有其他的失败。在RainStor，他们把这些测试标记为缺陷，但是他们仍然会执行这些测试以便进行进一步的功能回归检查。Adam Knight说：

“可能某天由于‘尾随零’导致测试失败了。如果第二天测试还是失败了，你可能就不想检查它了，因为你已经知道测试失败了。但是它也可能返回完全错误的计算结果。”

失败的回归测试包有个潜在的风险，就是它可能会在交付阶段晚期成为质量问题的“免费出狱”卡。请仅仅将已知失败的测试包作为临时空间使用，用来存放需要进一步查明的问题。无法及时发现问题是过程有潜在问题的警告标志。避免落入这种陷阱，有个较好的策略是限制加入已知失败回归测试包的测试数量。有些工具，比如Cucumber，甚至支持自动检查这样的限制。

创建单独的测试包去收集所有失败的测试，从项目管理的角度来说是好的，因为我们可以对其进行监控，并且在增长过快时采取措施。一两个并不重要的问题可能无须中止产品的发布，但是如果这个集合增长到几十个测试，那么就应该停下来做整顿，以免场面失控。如果有测试在已知失败回归测试包里放了很长时间，那就说明这个测试相关的功能可以去掉了，因为没有人关心这个功能。

10.3.2 自动检查那些被禁用的测试

适用于：失败测试被禁用，却没有挪到单独的集合中时

有些团队不会把失败的测试挪到单独的集合中，但是他们会禁用那些测试，这样已知的失败测试就不会再破坏整个验证过程。这种方法的问题是很容易遗忘这些被禁用的测试。

单独的测试包让我们能监控问题并确保它们最终被修复，或者我们在问题修复前无需再浪费时间去研究类似的问题。我们不能简单地把这些测试禁用。另外的问题是可能有人禁用了高优先级的失败测试，但他并不知道这个问题应该马上修复。

→如果你有禁用的测试，请自动监控它们。

爱荷华州助学贷款公司的团队有个自动化测试会检查哪些测试被禁用了。Tim Andersen说：

“人们禁用测试，是因为我们需要一个决定，或者我们在编写新的测试，不确定旧的测试需要如何进行调整。可能没有人再继续跟进讨

论了，或者大家只是忘记了重新启用这些测试。有时测试被禁用是因为大家还在编写这个测试，还没有具体的测试代码。

我们使用**FitNesse**寻找禁用的测试，我们有一个页面专门检查所有的测试名称。我们用它罗列出被有意禁用的测试，并在**JIRA**（缺陷跟踪系统）中增加一项记录。所以禁用列表相当于另一个测试。它必须与你想禁用的测试相匹配。在迭代最后，我们会查看这些禁用的测试。我们可能会说：‘这个测试不再有用，把它删了吧。’或者，‘哦，这里有个特殊情况，我们还没有从业务人员那里得到反馈。’在这些情况下，我们必须修复测试。”

如果你决定临时禁用失败的测试，而不是把它们挪到一个单独的集合中，那么请确保你可以1很容易地对它们进行监控，防止大家遗忘这些被禁用的测试。否则，你构建的活文档系统将会很快过时。禁用可执行需求说明是处理测试失败的一个快速而丑陋的临时解决方案，但这对于持续验证的整体来说是有益的。

一旦我们持续验证了需求说明，从功能角度判断系统在做什么就变得容易了，至少对于可执行需求说明覆盖到的那部分是这样的。然后需求说明就会成为描述功能的活文档系统。下一章，我将介绍如何通过演化文档系统最有效地使用可执行需求说明。

10.4 铭记

要频繁地验证可执行需求说明，以确保它们的可靠性。

相比用单元测试做持续集成，持续验证的两大难点是快速反馈和稳定性。

请建立独立的持续验证环境，并使部署完全自动化，使其更为可靠。

想办法获得更快的反馈。分开快速和缓慢的测试，为当前迭代的需求说明创建测试包，把运行时间较长的可执行需求说明集合拆分成更小的集合。

不要只是禁用失败的测试。要么修复问题，要么把测试挪到低优先级的回归测试包并密切进行监控。

第**11**章 演化出文档系统

我在第3章介绍了活文档的概念，并且解释了它的重要性，但还没有谈到构建它的方法。在本章，我将介绍很多团队实现活文档系统的做法。

存放可执行需求说明文件的目录并不能称为活文档系统。想要从活文档中受益，我们必须把需求说明组织得井井有条，不仅整体上好懂，而且要添加一些相关的上下文信息，使我们单独阅读某一部分时也很容易理解。

理想情况下，活文档系统必须有助于我们理解系统的行为，也就是说它提供的信息必须具有以下特点：

易于理解；

前后一致；

井井有条，便于使用。

本章我会介绍我调研的那些团队为达成这些目标所使用的方法。

11.1 活文档必须易于理解

如第8章所述，经过我们严格精炼建立起来的可执行需求说明具有专注性和不言自明的特点，并且使用项目的领域语言进行描述。随着活文档系统的增长，我们不断地为需求说明增添内容，也会将其进行合并或分解。在系统的增长过程中，以下建议有利于保持活文档易于理解。

11.1.1 不要创建冗长拖沓的需求说明

文档随着底层系统功能的增加而逐渐增长，你会创建新的需求说明，也会对现有的需求说明进行扩展。注意不要让需求说明变得太长。

→需求说明太长往往标志着有什么事情不对劲。需求说明越长越难以理解。

当需求说明太长时，可能会出现很多问题。例如：

所阐述概念的抽象级别不恰当。此时，可以问问你自己：“我们遗漏了什么”，并且试着找出遗漏的、可以帮助你分解测试的概念。找出遗漏的概念能带领我们找到设计的突破点。更多内容请参考7.5.2节。

需求说明描述了多个相似的功能，而没有专注在单个功能上。请将其拆分成多个单独的需求说明。详见8.3.13节。

使用脚本去描述功能，而没有使用需求说明。这时候，你可以重整相关内容，关注系统“要做什么”而非“如何做”。更多内容，请看8.3.2节。

需求说明包含大量不必要的上下文信息。可以对其进行清理，只关注那些描述特定测试的目标的重要特性。

11.1.2 不要使用许多小的需求说明来描述单个功能

随着系统的演进，我们对领域的理解也会发生变化。原先不同的概念到后来可能变得很类似——我们发现这是事物的两面性。类似地，我们可能会将复杂的概念分解成几个较小的概念，忽然之间却发现某个较小的概念与现有的某个概念很类似。在这些情况下，我们应该将活文档系统中描述同一功能的多个需求说明进行合并。

在天空网络服务部门，Rakesh Patel的团队在拆分需求说明时就一度做得有些过头了。单个需求说明不再描述一个完整的功能。Patel说：

“如果一个文件里含有大量的例子，该文件就会更加难以使用，因为大量类似的代码可能会混淆视听。以前我喜欢将不同的例子放在不同的文件中，但是后来我发现文件多了也会变得难以追踪。**”**

→如果我们必须看**10**份需求说明才能理解一个功能，那么是时候考虑重新整理文档了。

11.1.3 寻找更高层次的概念

在为系统增加功能的过程中，有时我们会发现有些类似的需求说明之间只有细微的差别。

→此时我们应该后退一步，从更高的抽象层次仔细去查看需求说明所描述的内容。

一旦我们找出更高层次的概念，一套完整的需求说明通常可以替换成只关注其不同之处的单个需求说明。这可以让信息更容易理解、寻找及使用。找出遗漏的概念还可能帮助我们找到系统设计的突破点，类似于7.5.2节所描述的过程。

11.1.4 避免在测试中使用技术上的自动化概念

适用于：项目干系人不是技术人员时

有些团队并不是拿可执行的需求说明来创建一种沟通工具，而是专注于利用它们去做功能回归测试以及编写技术上的验收测试。这样一来，开发人员编写测试更快了，但也使得测试人员阅读测试更加困难，对非开发人员来说更是无法理解的。Johannes Link在他使用FIT[①]的第一个项目里有过这样的经历：

“最终我们的测试有了大量的重复。开发人员能够理解测试，但是对业务人员来说这些测试就显得十分晦涩了。相比**JUnit**测试，维护与

运行这些测试的时间成本更高了。最后我们抛弃了这些测试中的一部分，并用**JUnit**进行了重写。”

注释：①第一个专用于可执行需求说明的自动化工具。

→如果我们想把需求说明当成一种沟通工具，那么以技术语言来描述需求说明，是一种比较低效的做法。如果商业用户关心可执行需求说明，那么需求说明就应该用他们能理解的语言来描述。如果商业用户并不关心，那么我们就应该使用技术化的测试工具来获得需求说明。

当活文档系统包含技术上的自动化概念时，比如为了确保某个处理过程的结束而等待特定时间的命令，就表明团队需要重新审视底层系统的设计了。在活文档中需要使用技术化的概念，往往意味着系统设计出了问题，比如异步进程的可靠性出了问题（详见本章末尾的11.4节）。

可以使用技术语言的唯一情况是利益相关者也是技术人员，而且他们能以技术语言(比如SQL查询、DOM的标识符等)理解在做什么事情时。请注意，即使利益相关者是技术人员，我们在使用此类技术语言时往往还是会描述“如何测试某个功能”，而不是系统“具有什么功能”。虽然编写此类测试可能一开始比较快，但是长远来看它们会导致维护问题。更多内容请参见8.3.2节。

为了与能理解脚本的利益相关者一起描述项目的验收测试，Mike Vogel使用了DbFit，这是我编写的一个FitNesse扩展，用于编写数据库测试脚本。事后看来，他认为这是个错误：

“一开始他们很高兴，因为他们可以快速地使用**DbFit**而无需编写定制的**fixture**，这样，他们从第一天开始就拥有了自动化测试。后来，随着项目复杂度的增加，发布计划时间趋紧，想要回去重新创建测试**fixture**，让系统更易测试并且让测试更为简单、更易理解，已经是可望而不可及的事情了。最终这种做法让我们的测试很脆弱很复杂。”

11.2 活文档必须前后一致

活文档可能是项目里最“长寿”的工件了。技术会推陈出新，代码会新旧更替，而活文档系统描述的是业务如何运作。我们会在几个月或者几年内不断为其增加内容，并且需要让它在以后查阅时能让人理解。对许多团队来说，其中最大的一个挑战是让活文档的结构和语言保持一致。Stuart Taylor对此进行了巧妙的解释：

“如果你编写的**BDD**测试（可执行需求说明）非常清晰，比方说由

于每次使用的方式不同，跳转到一个页面会有**57**种方法，而你的测试详细描述了这些方法，那么这其实是一个危险信号。不断地重构描述语言，使其既不要抽象到难处理的程度，又不要详尽到不像是**BDD**测试，这点是很重要的。”

前后一致的语言也可以让可执行需求说明的自动化过程更加高效。对Gaspar Nagy来说，这是开发中的一个关键原则：

“对验收标准来说，使用一致的措辞与一致的表达语言很重要。这样开发人员更容易确定它是否与现有的结构一样，并且更容易自动化。”

想要保持活文档的一致性，我们必须不断地对其进行完善，使其与当前软件系统应用的模型保持同步。随着软件概念的不断演进，活文档系统中也需要相应地体现出来。这些维护需要成本，但若非如此，也就不存在活的文档了。

11.2.1 演化出一种语言

几乎所有团队最终都演化出了一种需求说明的语言，一组需求说明可重用的模式。对于某些团队，这种语言演化了几个月，并且往往背负着维护性问题的“罪名”。

Andrew Jackman说BNP Paribas公司的Sierra团队是在发现自动化层增长过快的情况下开始演化这种语言的：

“我们现在有非常多的**fixture**代码，逐渐出现了维护性问题。我们有大量非常特定的**fixture**，它们冗长的描述只在一个测试里用到，我们试图将其浓缩成一种通用语言。我们在**FIT**里开发了一种给**web**测试使用的迷你型领域特定语言。这样就减少了大量的**fixture**代码。”

有些团队很快就演化出了这种语言的基础。Rob Park所在的属于一家美国大型保险机构的团队就是一个很好的例子：

“语言演化得很快。最初的三四个**fixture**［自动化的测试**]**类都是相互独立的。我们试着让它运行起来并且专注在对话描述部分。我们立刻注意到**step**文件（自动化测试的类）里有些重复，然后我们开始将重复部分从中剥离。每个用户故事都有一个**Gherkin**①文件（可执行需求说明），但是我们描述同一个功能的故事卡有五六张。

这些用户故事的**step**都非常相像，所以我们发现即使描述一个业务功能的用户故事有好几个，使用单个**step**文件的实际效果也会更好些。否则，即使故事的描述只有一行，也会出现大量的重复。”

注释：①Gherkin是Cucumber和SpecFlow里用于描述功能的语法。——译者注

→演化出一种语言有助于降低自动化层的维护成本，因为重用现有的短语描述新的需求说明，可以促成需求说明的一致性。

只要活文档系统是自动化的，并且直接与软件相连接，那么它就可以确保软件模型与业务模型相一致。因此，为活文档系统演化出一种语言是创建并维护统一语言（如第8章所述）的一种绝佳方式。

11.2.2 将需求说明语言拟人化

适用于：**Web**项目

有些团队通过虚构人物来描述用户故事，特别是在开发网站的时候。在这些情况下，需求说明语言可以从不同人物能够执行的操作上来提取。

→虚构人物有助于简化可执行需求说明并且可以使其更加容易维护。

爱荷华州助学贷款公司的团队将其需求说明里使用的语言进行了拟人化。Tim Andersen说：

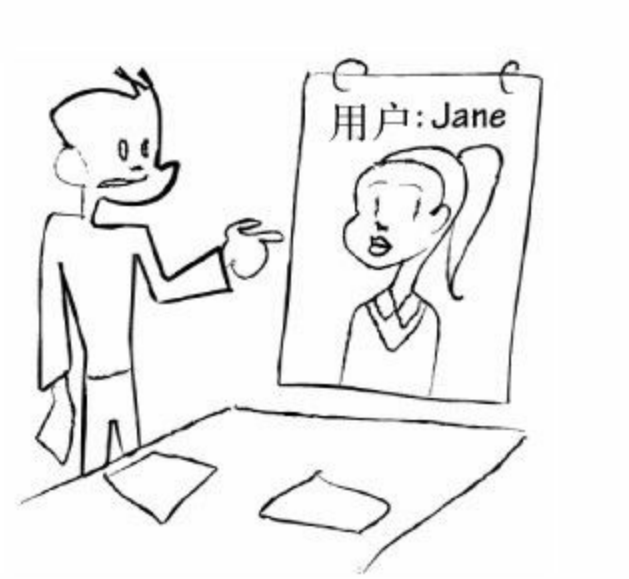
“我们并不会模糊地使用用户的概念，我们使用的是不同的人物，并考虑他们使用系统的动机、方式及其目的。我们还为不同的角色取了名字。**Boris**是借款人，**Carrie**是共同签署人。

虚构人物很有帮助，因为它们可以让我们从一个使用者的角度来考虑系统的行为。并且它们还有很多我们事先没有预料到的其他作用。例如，因为虚构人物是测试助手（自动化组件），它可以从一个更舍适的切入点与系统进行交互。**”**

对于用户交互不多的项目来说，使用虚构人物并没有太大的意义。因为在之前的项目里使用虚构人物取得了成功，Andersen曾尝试将相同的概念应用到一个技术性的数据处理系统中。他最终放弃了这一做法，并改用了过程流模型。

“多个不同来源的数据载入到电话系统，这样人们才可以打电话。电话数据更新后，我们会将其发回给数据提供方。这是一个批处理过程。事实上没有真正的使用者，它只是自己在运行而已。虚构人物并不适舍。我们尝试使用虚构人物来定义测试，却没有得到业务人员的认同。所以我删除了所有虚构人物的代码，并且改成了基于使用**Given-When-Then**关键字的过程。这样一来清爽多了，对每个人来说也更容易接受。**”**

围绕着虚构人物的行为来演化统一语言，可以确保我们对每个人物在系统中有哪些需求的理解，与他们实际如何使用系统的行为是相一致的。这可以推进需求说明所使用的语言与结构，并可以帮助我们打造前后一致的文档系统。



11.2.3 协作定义语言

适用于：不进行需求说明工作坊时

→如果你决定不进行大型工作坊而使用其他替代方法，那么请协作定义统一语言。

Christian Hassa说协作定义统一语言是他们团队面临的一个最大挑战：

“要构建一种统一的、可以良好绑定的领域语言，如果没有任何指导是完全不可能做到的。对测试人员编写的内容，开发人员不得不重新进行梳理。有时候这是因为测试人员记录东西所采用的方式不够明朗或者不容易绑定（自动化）。测试人员编写了多少内容，我们就得重新梳理多少内容。如果我们在他们编写第一个例子的时候就立刻进行绑定（自动化），那么我们就能及时发现问题。

这就像结对编程与事后的代码评审相比较。在结对编程的时候，如果对方觉得你做的不对，他会立刻告诉你，这样可以及时改正。而对于代码评审，如果你发现了问题，你可能会说：好的，下次我会换一种做法，这次就先这样吧。”

Christian Hassa建议，与其在发现不一致性问题后回头去修正，不如让开发人员与测试人员结对编写需求说明，这样可以预防此类问题的出现。这就好比飞机的驾驶员与副驾驶一起进行协作，以便预防问题的出现。这可以极度降低需求说明质量低下的风险，因为在编写过程中另外一个人验证了需求说明，并且提防着问题出现。

BNP Paribas公司的Sierra团队有一种比较稳定的语言，这种语言演

化了很多年，他们的业务分析师可以独自使用这种语言编写新的需求说明。为了避免语言不一致或者需求说明难以自动化，他们会要求开发人员评审那些与现有需求说明在结构上差别很大的需求说明。Bekk咨询公司团队在挪威奶牛记录系统项目中使用了类似的过程。他们的业务人员会编写带例子的需求说明，而开发人员会对此进行评审，并且提供建议使其与活文档系统的其余部分更加一致。

11.2.4 将构建模块文档化

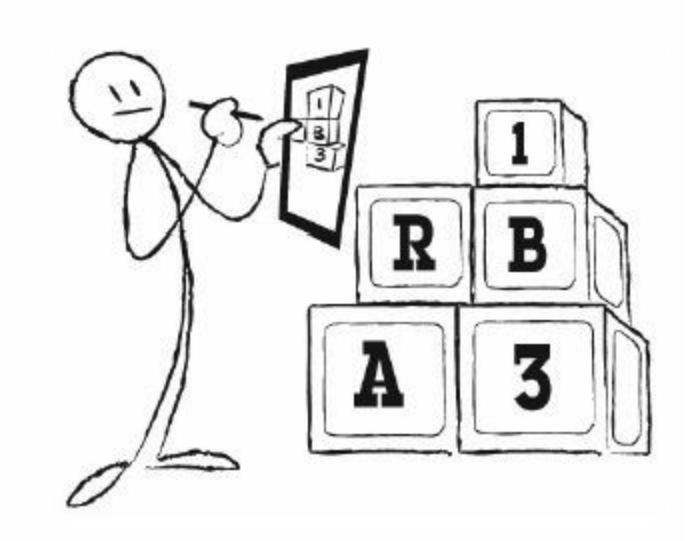
→将需求说明的构建模块文档化是一种很好的做法。这有助于我们重用组件并维持语言的一致性。

有些团队为构建模块划分出单独的文档区域。在爱荷华州助学贷款公司，他们有一个页面包含了所有的虚构人物。其中没有任何断言，但是它却展现了哪些构建模块已经存在。这个页面是由底层的自动化代码生成的，它为活文档创建了活字典。

为项目语言构建优秀的文档还有一种更为简单的方式。在为编写本书进行调研的过程中，当参与者被问到对于团队的新成员，他们会给予什么样的建议，以便编写出良好的需求说明时，几乎所有人都建议去参考现有的需求说明。为了将需求说明的构建模块文档化，从现有的需求说明中选出一些好的、具有代表性的例子，不失为一种好的做法。因为这些需求说明已经是可执行的了，所以为构建模块编写的文档，其精确性与一致性是有保证的。

因为在团队构建项目的长期过程中，活文档始终为团队提供支持，所以可能存在一些文档还在使用过时术语的风险。有人可能会使用团队三年前开始使用的术语，而有的则可能会使用两年前才开始使用的术语，等等，这取决于需求说明最初是什么时候编写的。这几乎完全违背了活文档系统的观点，因为我们需要将老的术语翻译成新的术语。

在语言演化的过程中，维持整个文档系统的一致性并不需要耗费多少精力。而且一致的文档将会给团队带来更多的长期价值。



11.3 活文档必须组织得井井有条，便于访问

活文档系统增长迅速。随着项目的推进，开发团队会频繁地加入新的需求说明。需求说明在项目进行几个月后达到数百个是很常见的。我采访过几个团队，他们在几年时间里对活文档系统的签入次数超过了5万次。

要让活文档发挥作用，用户必须可以很容易地找出所需功能的描述，这意味着整个文档系统必须组织有序，并且单个需求说明必须便于访问。

Phil cowans说，他对于活文档最大的经验是团队应该尽早考虑高层次的组织结构：

“我们没有考虑测试的高层次组织结构，只是不停地添加新的测试，结果就是很难找出哪些测试涉及了哪个功能。收集有关网站功能系列的描述并据此（而不仅仅是单个功能）组织测试会有所帮助。我认为这样有助于开发出比较容易理解的产品和易于维护的代码库。”

如果每次我们想要理解某个功能如何运作，就不得不耗费数小时从貌似无关的数百个文件里拼凑出全局，那么很可能我们还会去阅读程序代码。为了尽可能发挥活文档的优势，信息必须很容易查找到。以下是一些相关的技巧。

11.3.1 按用户故事组织当前的工作

很多可执行需求说明的自动化工具允许我们将需求说明按层次结构归类，要么是以网站节点与子节点的形式，要么是以文件目录与子目录的形式。

→如果你使用可执行需求说明的自动化工具，那么将目前正在进

行的工作归类在一起通常是一种很好的做法。

正如10.2.6节所建议的，按层次结构归类需求说明，我们就可以将此类需求说明以测试包的形式快速执行。

通常一个用户故事需要我们更改多处功能区。例如，一个关于改进注册功能的用户故事，它可能影响后端的用户报表以及系统验证年龄的行为。也可能需要实现与PayPal和Gmail的新的集1成。所有这些功能都应该由独立的并且专门的需求说明来描述。同时对于每个用户故事的时间完成标准还要有清晰的定义。与一个用户故事相关的所有内容都应该归类在一起，以便容易地执行所有这些测试。

请看图11-1所建议的组织方式：“当前迭代”的分支。

11.3.2 按功能区域组织用户故事

用户故事作为计划的工具是非常出色的，但是它在组织现有系统功能方面并没有太大用处。如图11-1所示的编号为128的用户故事，其中有一部分是PayPal集成，当做完这个用户故事半年后再回头来看，编号128的用户故事与PayPal集成就显得风马牛不相及了（除非要追踪它的来龙去脉）。如果有谁想要知道PayPal的集成原理，他需要知道具体的用户故事编号才能找到。

→很多团队在完成可执行需求说明后，会按功能区域对其进行重新组织，使其具有层次结构。这样浏览基于业务功能的层次结构就可以很容易地找到一个功能的详细解释。

图11-1中，“功能集”下的分支展示了这一点。一旦编号128的用户故事完成后，我们应该将PayPal集成的需求说明转移到“付款”分支之下，将“后端用户报表”改成“用户管理”，等等。以这种方式组织活文档系统后，如果需要讨论MasterCard付款功能的需求变更，我们就可以很快找到与此相关的所有例子。

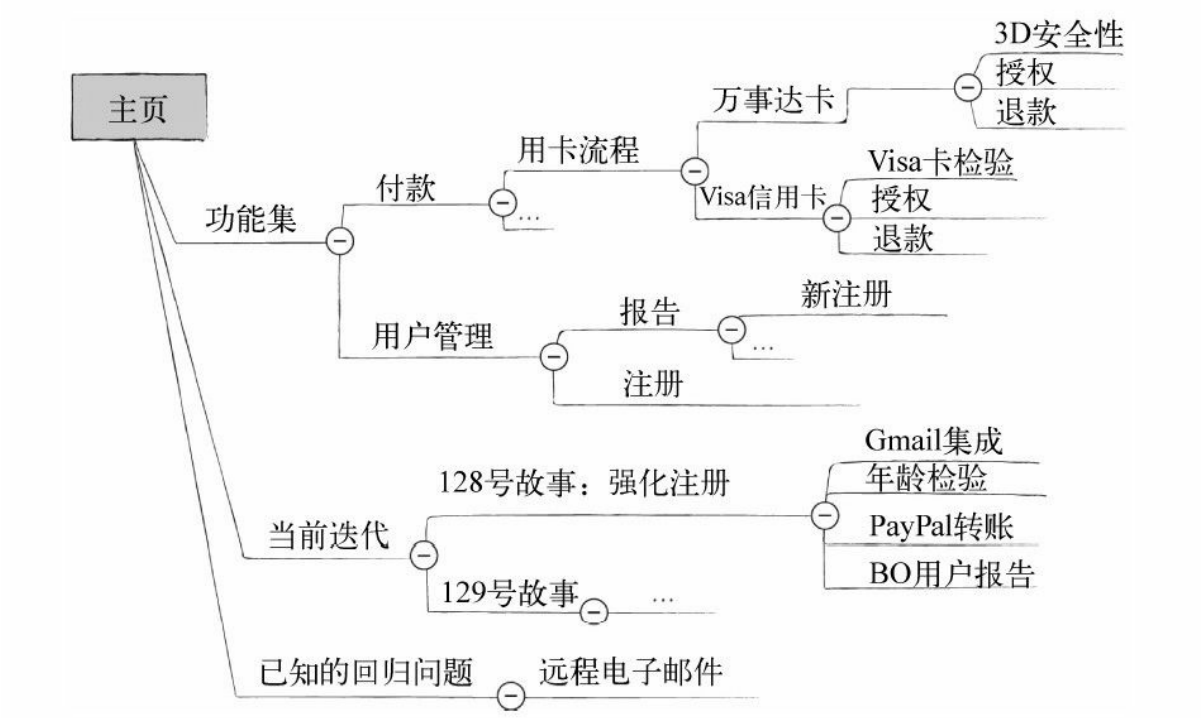


图11-1 以功能区域（比如付款和用户管理）组织的活文档层次结构。当前迭代的需求说明是以用户故事和功能来组织的。一些还需等待更多信息的已知问题，也可以单独放在一个分支里

如果还想要知道某个功能具体是由哪个用户故事引入的，还有一些工具可以让你以不同的层次结构来交叉引用同一个需求说明。

11.3.3 按用户界面的导航路径组织

适用于：用户界面的文档系统

→在活文档系统中复制用户界面的导航结构。

Beazley公司Ian Cooper的团队在他们的活文档系统里应用了一种创新的组织方式。他们没有按照功能区域来组织，而是在活文档系统里复制了用户界面导航结构。Cooper说：

“FitNesse测试让我们可以找出用户故事的相关内容。但是使用这种方式来浏览却异常艰难。怎样才能找到代表软件某一部分功能的用户故事呢？

我们对此进行了重组，这样**FitNesse**页面看起来像是一份帮助页面。当我打开某个页面时，有**FitNesse**测试可以告诉我本页面具有哪些功能。如果我点击对话框旁边的某个链接，它会跳转到另外一个页面，那个页面对该对话框有详细解释。这样如果想要了解什么功能，很容易就知道在哪里可以获得这类信息。**”**

对于那些清晰地定义了导航路径的系统来说，这种方式很直观，比如办公系统。但是如果用户界面导航路径会经常变换，那么这可能会导致很多维护性问题。

11.3.4 按业务流程来组织

适用于：端到端的用例需具备可追溯性时

→按照业务流程来组织活文档系统，在端到端的用例里可以很容易地追溯系统的功能。

Mike Vogel参与了一个为药物研究提供支持的软件系统，他的团队按照业务流程来组织他们的活文档系统。他是这么解释这个方法的：

“我们按照用例来组织**(FitNesse)**测试。我们的用例是按层次结构组织的，顶层的用例代表系统的目标。每一个顶层用例也是对端到端业务流程目标的定义。如果一个用例会引用一些较低层级的用例，那么它就是子流程。

需求文档的目录与我们的测试目录是一样的。这让我们可以更容易地理解测试如何与业务流程相一致。同时它还建立了从业务需求到测试的直接可溯性，这很关键，这样才能满足我们领域里的管理需求。**”**

这些并不是组织活文档仅有的方式。另外一个很好的方法是按照帮助系统或者用户指南的章节来组织信息。请将这些想法当作灵感，触发你为自己的文档系统找到建立层次结构的最佳方式。

11.3.5 引用可执行需求说明时请使用标签而不要使用URL

适用于：需要需求说明的可追溯性时

如今，许多活文档工具都支持标签功能——自由格式的文本属性，可以附加到任何页面或文件上。对于可追溯性来说，相对于使用用户故事或用例来保持需求说明的层次结构，使用这类元数据通常效果更好。当领域模型变更时，活文档应该也要作出相应的变更。需求说明往往会迁移、合并、分解或者变更。想依赖于严格的静态层次结构来获得可追溯性是不可能的。但是，如果我们将故事或用例的编号当作标签附到需求说明上，就很容易实现可追溯性。

如果你想要在其他工具中（例如缺陷追踪系统或进度计划工具）引用活文档页面，那么标签也会非常有用。如果我们在引用可执行的需求说明时使用基于页面当前所在位置的URL，那么该需求说明以后将无法随意挪动，否则URL链接将会失效。

→附上标签，并且链接到相应的搜索结果，可以让系统更具弹性，方便将来进行改动。

即使你用的工具不是基于Web的，而是将需求说明放在项目的目录

里，只要有一个简单的脚本，你还是可以使用标签。Bekk咨询公司的挪威奶牛记录系统团队就是这么做的。Børge Lotre解释道：

“我们使用**Confluence**与客户共享**Cucumber**测试，并且直接将**Cucumber**测试从**Subversion**里链接到**Confluence**。这样我们重新组织**Cucumber**测试的文件结构就会有麻烦，但是使用标签可以帮助我们克服这个缺点。现在，我们使用标签来记录哪些**Cucumber**测试覆盖了哪些需求。”

不要直接引用活文档系统中某个具体的需求说明，因为这会妨碍我们以后对文档进行重组。将可以动态搜索的元数据、标签或者关键词作为外部链接要好得多。

活文档系统不仅仅是一堆可执行的需求说明。深埋在杂乱不堪的测试里作为文档的信息是毫无用处的。为了获得实例化需求说明的长远利益，我们必须确保文档的组织方式有利于大家快速地找到某个功能的需求说明，并测试该功能。

我在这里介绍了需求说明最常见的组织方式，但是你可以继续挖掘。找出适合你自己的方式来组织文档，能够方便商业用户、测试人员以及开发人员直观地找到他们想要的内容。

11.4 聆听活文档

最初，许多团队不明白活文档实时反映了它所描述系统的领域模型。如果系统的设计是由可执行需求说明驱动的，那么需求说明与软件将会用到相同的统一语言和领域模型。

在可执行需求说明里发现偶然的复杂性，意味着你应该去简化系统并使其易于维护和使用。Channing Walton将这种方法称为“聆听测试”。他在UBS开发订单管理系统，其工作流的验收条件非常复杂。他说：

“如果测试太复杂，就说明系统有些问题。工作流的测试非常痛苦。有很多事情要做，并且测试非常复杂。开发人员开始质疑为何测试如此复杂。结果发现由于每个部门并不知道其他部门的工作内容，导致工作流太过复杂化。测试有助于大家看到其他部门也在做验证并处理错误，因为它将所有内容放在了一起。调整之后，整件事情简单多了。”

自动化可执行需求说明强迫开发人员使用自己开发的系统，体验其使用感受，因为他们必须使用为客户设计的界面。如果可执行需求说明难以自动化，那就代表客户端API很难使用，也就意味着是时候简化API了。这就是Pascal Mestdach获得的一个最重要的经验教训：

“你编写测试的方式就确定了你如何设计并编写代码。如果你需要

在一部分测试中持久化病人的数据，你就需要构造一个数据集，将**4**个表格的数据填入数据集，然后调用一个大型方法进行持久化，并且调用该类的一些配置方法。这让真正的场景测试难以进行。如果配置很难，那么测试也会很难，那么在真实代码中持久化病人的数据也将变得比较困难。”

Markus Gärtner指出，冗长的配置标志着糟糕的API设计：

“当你注意到一串冗长的配置时，请为你的**API**用户以及你所构造的东西着想一下。处理你复杂的**API**将让某人痛苦不堪。你真的打算这么做吗？”

难以维护的活文档也暗示了系统架构并不理想。Ian Cooper说领域代码的微小改动经常导致很多活文档系统的测试失败，这就是“霰弹式修改”的一个例子。这促使他去研究如何改善系统的设计：

“这意味着架构有问题。起初你纠结于此，而后你开始意识到问题并不在**FitNesse**身上，而是它与系统交互的方式有问题。”

Cooper建议将活文档看作是系统的另一个用户界面。如果这个界面难以编写和维护，那么真实的用户界面也将遭遇同样的问题。

在活文档里，如果有一个概念是通过复杂的交互来定义的，那么很可能意味着程序代码里也存在同样复杂的交互。如果活文档里描述了两个类似的概念，那么很可能领域模型里也存在同样的重复。不要无视复杂的需求说明，反之，可以将其当作一个警示标志，督促我们去变更领域模型或者清理底层的软件系统。

11.5 铭记

为了尽可能发挥活文档系统的优势，请保持其一致性，并确保单个可执行需求说明易于理解，并且任何人都能方便地访问，包括商业用户在内。

演化并使用统一语言，保持一致性。

随着系统的演进，请注意冗长的需求说明，或者几个小的只有细微差别的需求说明。寻1找更高抽象级别的概念会让这些事情更容易阐释。

将活文档系统组织成层次化结构，可以很容易地找出当前迭代的所有需求说明，以及之前实现的所有功能。

Part 3 第三部分 案例研究

本部分内容
第12章 uSwitch
第13章 RainStor
第14章 爱荷华州助学贷款公司
第15章 Sabre Airline Solutions
第16章 ePlan Services
第17章 Songkick
第18章 思想总结
附录A 资源

第12章 uSwitch

uSwitch.com是英国最繁忙的网站之一。网站比较不同公司及产品的价格和服务，包括能源供应商、信用卡、保险公司。他们软件系统的高扩展性以及与大量外部系统的复杂集成导致了很高的复杂度。

uSwitch是个有趣的案例研究，因为它展示了一个拥有独立的开发团队和测试团队、按瀑布流程工作的公司，如何依然能在问题颇多的遗留环境中转换到一种更好的方式，去交付高质量软件。uSwitch在3年的过程中彻底地改变了他们的软件交付流程。

在uSwitch，我采访了Stephen Lloyd、Tony To、Damon Morgan、Jon Neale和Hemal Kuntawala。当我询问他们的软件流程时，他们的回答基本上就是：“有人早上提出的一个想法，当天就可实现并上线。”我早年所在的一家公司的软件流程也是如此，但是当时几乎每天都会在生产系统中碰到许多问题。而在uSwitch，产品的质量和交付功能的速度都很令人羡慕。

虽然uSwitch没有特意实施实例化需求说明，但他们现在的流程包含了本书所描述的几个最重要的模式，包括从目标获取范围、协作制定需求说明，以及自动化可执行需求说明。为了改进软件开发流程，他们经常寻找并解决妨碍提高质量的问题，从而专注于提高产品质量。

在寻找更好的方式使开发与测试相一致的时候，他们以让人易读懂的形式实现了测试的自动化。之后，他们发现测试可以当作需求说明。使用可执行需求说明让他们能更好地合作。在提炼流程的过程中，他们不得不让持续验证更可靠，这又帮助他们提高了提炼需求说明的方式。

当他们去寻找与商业用户合作的更好方式时，他们开始从目标获取范围。

12.1 开始改变流程

2007年，uSwitch使用的是瀑布开发流程，大型项目都是预先做大量的设计。新的CTO在2008年推动团队转型敏捷，他们引入了Scrum，3周一个迭代。2008年10月，uSwtich新功能进入市场的平均时间为6～9周。虽然对瀑布流程来说，这是一个巨大的进步，但是每个sprint还包含了大约40%的未计划的工作。Scrum最好应用在跨职能团队上，但是由于他们的开发的组织方式，他们并未真正做到这一点。

QA团队和开发团队是分开的。因为测试人员使用QTP（开发人员无法获得相关信息），开发人员开发和测试人员测试都没有互相沟通过。结果，开发人员发现很难知道何时才是真正的完成。因为发布的标准是所有的QTP测试必须通过，测试往往是流程的瓶颈。

sprint最后的部署流程平均需要3天，这主要是由于测试，并且还存在很多质量问题。当团队使用短迭代的时候，QTP测试开始需要大量的维护。Hemal Kuntawala说：“没人知道他们做了什么，他们完全是在浪费时间。”

这致使全公司的精力开始专注于质量，每个人都被要求开始考虑质量问题。他们认识到开发人员把东西交给测试人员而没有任何解释，这是个问题，于是决定合并测试团队和开发团队。他们去掉了不同的职称；测试人员成为有特殊专长的“开发人员”。同样，需要专业数据库知识的任务交给了有经验的数据库开发人员，测试人员则认领需要专业测试知识的任务。但是他们不再是独自负责测试了。程序员开始寻找更好的方式来编写单元测试和功能测试。团队决定使用Selenium，而不是QTP，这样使测试更轻量，大家也都能访问到。这帮助了开发人员和测试人员更好地合作，但是因为Selenium太技术相关了，这样的变化没有带来和商业用户更好的沟通方式。

因为uSwitch没有任何可靠的系统文档，而这个系统已经花了10年时间开发，所以遗留的商业规则常常导致理解问题。Kuntawala说：

“有一天，我们遇到一条能源（子系统）上的遗留商业规则，而我完全不了解，这很令人沮丧。我需要一种方式让我们不需要看代码和单元测试就能了解商业规则以及应用程序的工作原理。而且单元测试并没有做到覆盖所有的东西。**Google**一番后，我们发现**Cucumber**能弥舍测试与描述目标功能之间的差异。我们可以用易懂的语言来描述我们需要什么，而它能适舍于开发人员想达到的从外到内的方式。**”**

为了让大家熟悉新工具，他们开始把Selenium测试转到Cucmber。这么做仍然是测试自动化，也就是功能完成之后再检查，但已经朝着测试先行的可执行需求说明迈出了第一步。Jon Neale解释道：

“Given-When-Then格式的**Cucumber**测试迫使我们重写了故事，明确我们在做什么，并让我们看到我们已经遗忘的东西。”

团队开始让商业利益相关者了解不同的Cucumber场景，不仅仅为了验证边缘情况，而且还为了指出哪些场景比较重要，以此减少范围，避免“以防万一”的代码。

在他们完成从Selenium测试到Cucumber的转换并和商业用户一起审查过后，他们发现在迭代最后做测试完全没有意义。Neale说：

“我们意识到可以坐下来通过需求说明工作坊来获得许多东西，清晰地勾勒出我们要达到什么目的以及如何才能达到。”

接着团队引入了需求说明工作坊，用这种方法来和商业用户合作，明确未来需求的验收标准。这显著改善了组内的沟通。开发人员（此时测试人员也称作开发人员）开始了解领域知识。商业用户了解边缘情况以及较模糊的用户使用场景，因为开发人员会问及此事。

这种改变还影响了工作的分割。之前，工作大多按照技术任务划分。用那样的技术分块，他们发现很难为每个任务写出具体的验收标准。改变后团队的注意力从完成任务转移到实现功能应当交付的价值上，他们开始从用户的角度描述故事，这使得讨论并说明各工作块的验收标准变得更容易了。

新的组织结构还能让他们更频繁地发布软件。因为技术任务会互相依赖，开发人员在一大块工作完成前不会去部署。通过专注于用户故事，他们的工作块更小、更独立，并且可以更频繁地发布。

12.2 优化流程

随着可执行需求说明数目的增加，团队发现测试结果不可靠。即使系统中的功能是正确的，环境的问题也常常导致测试失败。他们运行测试的环境不稳定。他们有开发环境，测试环境和临时环境，但是没有一个适合频繁运行可执行需求说明。

因为开发人员使用开发环境做各种尝试，所以这个环境经常有问题。测试环境用于手动测试，按需部署。两次部署间会有多个改变，所以当测试失败时，不知道是什么导致了问题。商业用户也在这个环境中进行手工测试，这会影响自动化Cucumber的测试结果。临时环境是生产环境的镜像，用于最终部署的测试。

uSwitch又创建了一个环境，只用于持续验证。这解决了稳定性问

题：有专门的环境用于测试，而不会打扰其它工作。这个环境由他们的持续构建系统自动部署。用这个环境，可以更快地得到可执行需求说明的反馈，反馈结果也明显可靠很多。

一旦解决了环境问题，排除了这个不稳定因素，他们就能够看到哪些测试或软件的哪部分由于设计而导致了不稳定。因为所有的测试是通过用户界面执行的，运行测试的可执行需求说明数目的增长会导致瓶颈。有些测试很慢，有些测试不可靠。他们开始去除不可靠的测试，查明导致不稳定的原因并做改进。测试是在技术层面上实现的，因此带来了维护问题。

团队开始重写测试、拆分，并提升抽象层次。Kuntawala说这对他们来说是迈出的相当大的一步：

“当我们刚开始编写测试的时候，会依赖于浏览器专有的东西，比如，页面上的**DOM**标识符，但是这是会变的。我们习惯了**Cucumber**的语法与功能后，开始用真实的商业语言编写测试。之前你可能会说：**‘**用户在某某**_id**的输入框里输入数字**100**。**’**现在则是：**‘**用户输入有效金额。**’**有效金额会在单独的测试中定义。编写了这个测试后，你就不必在其他每一个测试中显式地进行测试了。有效金额的测试可能尝试负数、字母等，但是这些都从所有其他测试中抽象出来。这是很大的一个进步。**”**

为了降低长期维护成本，uSwitch团队开始提炼需求说明，演化一种供需求说明使用的一致语言，寻找缺失的概念，以提升抽象层次。

可执行需求说明有了相对较好的功能覆盖，加上稳定的持续验证环境，uSwitch团队对他们的代码有了更多的信心。但是他们的测试运行还是很缓慢，没有带来他们所期望的快速反馈。他们觉得不是每个测试都值得做自动回归检查。有些测试在开发过程中很有好处，但是不会给功能带来好处。

有个例子：发送延误的邮件。他们在实现功能的时候自动运行可执行需求说明，但是功能开发完成后就禁用这些测试。这样低风险的测试不会在持续验证流程中运行。这给了他们更快的反馈，并且减少了测试维护成本。下次有人开发系统这部分的时候，他们又会重新启用测试，并按需进行清理。

运行测试或验证系统是否能上线不再是瓶颈，现在部署到生产环境成了流程中最慢的部分。开发人员和运营工程师结对一起分析什么拖慢了速度。最后发现是部署前的测试造成了拖延。有些测试在临时环境超时，这些需要运营工程师重新运行全部的测试。通过找出环境的不同之处并重写测试，使它们更灵活，开发人员将整个测试的执行时间从2小时减少到了15分钟左右。

结对还帮助运营工程师融入到了流程中。之前，他们会报告测试失败了，但是他们的报告缺少细节。一旦他们理解了如何解释测试结果，出问题的时候就能给开发人员提供更有意义的报告。

接下来的改变是让商业用户更多地加入到开发过程中来。虽然此时团队使用用户故事作计划，但是用户故事是他们自己编写的。商业用户开始和开发团队一起编写故事，对需求说明承担起更多的责任。他们先大概地定义价值（用户故事中的“为了”），然后开发人员定义解决方案（用户故事中的“我要”）。商业用户还开始负责举办需求说明工作坊。这改善了团队的沟通。Damon Morgan解释道：

“他们以前脱离了流程。他们会问：‘这个功能我们可以有吗？’，然后我们用某种他们不一定能理解的奇怪语言记录下来。他们会看到这个任务在板子上挪动，但这对他们来说没有任何意义。在我们开始举行需求说明（工作坊）并和他们深入讨论具体交付的细节，得到这些故事可执行的标准并和他们一起编写故事后，他们对整项工作的主人翁意识增加了。故事从商业用户那里返回来，他们不会再说：‘这个功能不对’，而更多地会说：‘我们还没有全团队一起考虑这个场景。’”

有了商业用户更多的参与，uSwitch的团队建立了信任和信心。这意味着不再需要长期优先级排序以及大块的工作。这也意味着商业用户会更容易接受开发团队的建议。

有了更紧密的合作和更多的信任，商业用户提出的开发范围也不同了。团队接着开始把需要的功能分解成了一些可发布的小块功能，这些功能仍然能为业务带来价值。

有这样一个例子，重写能源目录的流程是4级页面，包含能源供应商和计划的列表。他们不是一次性将它们全部发布，而是一次重写一个页面，把页面和其它服务关联起来，然后发布。虽然这种方式提高了集成成本（因为新页面要和老页面集成），但是由于发布更早，所以他们获益不小。重写目录的一个原因是为了搜索引擎优化：一次发布一个页面意味着Google可以更早地索引页面。团队还发现，更小的发布可以降低出错的风险。如果出了问题，可以标记成一个特殊的发布。更小的发布更容易查明问题的原因。

当团队在一个迭代内有多个交付时，在迭代最后签收就变成了瓶颈。他们不是在最后做一个大的演示，而是在功能的一个可发布的部分完成时就把新功能演示给商业用户并签收。

团队发现不再需要正式的需求说明工作坊，取而代之的是非正式的聊天环节。做小块的工作、获得快速的反馈让团队在有足够的信息开始工作时能够提速，甚至他们不必获得完成任务的足够信息。根据Damon Morgan的说法：

“一开始，（需求说明工作坊）会议很长很大，我们尝试研究大量的需求。现在真的是‘我们现在就开始实现这个功能’，并且是相对较小的功能，因此我们会和相关人员交谈。整个团队会一起举行类似于迷你需求工作坊的东西，但是真的只是谈话，甚至不需要会议室。你获得了验收条件，就开始着手工作，并且更快地展示给用户。通常两天就可以完成并交付，然后开始下一个。我们在一个迭代内可以这样重复多次。”

因为流程让开发人员比以前学到更多的业务领域知识，并且他们不再因为误解业务需求而导致很多问题，所以他们不需要事先知道很多就可以正确地完成工作。Stephne Lloyd说：

“作为一个团队，我们比以前更好地在一起舍作，并能更好地理解业务需求。一字不差地记录他们的需求就显得没那么重要了，因为我们现在对领域的理解比一年前要好得多。”

最终，uSwitch的团队开始按需部署，完全不受迭代的限制。为了更好地演化这个流程，他们开始经常监控他们的生产系统，追踪错误率和新功能的使用情况。这些附加的可见性为被忽视的部署问题建立了一张安全网。

12.3 当前的流程

经历了这些改变后，开发流程变得简单多了。现在是轻量的，并且是基于流而不是基于迭代。

每日站立会议上只要有人提出新的想法，就会进入待办列表。任何人都可以提出新的想法，包括商业用户和开发人员。新想法主要是在站立会议上讨论并排出优先级。提出想法的人为了更好地解释他的想法，可能会在会前画一些粗略的图表，或准备业务案例。此外，除了需要和外部合作伙伴签订合同，一般不需要大量的预先准备。

当有个故事获得最高优先级时，团队会考虑完成这个故事需要哪些步骤。所有对该故事有兴趣的人会碰到一起并简要地讨论一下到底需要什么，并写下验收标准。过去，团队尝试在这些会议上编写Cucumber测试，但是他们确定做Cucumber测试所用语法时，使用的方式是这样的：一个人敲键盘，其他人干看着。这造成了讨论的中断。

开发团队、市场部人员以及负责Email功能的团队大家的座位离得很近，所以他们不需要预先知道许多细节就可以开始工作。开发人员会开始接手这个用户故事，频繁地与商业用户交谈，询问更多的信息或重新审视验收标准。

验收标准在开发过程中转化成Cucumber测试并自动化。开发人员使

用探索测试来在修改前理解系统的现有部分。有时他们使用客户会话日志来理解真实用户如何与网站某个特定功能进行交互。基于此，他们开发Cucumber测试并在开发过程中捕获需要考虑的用户使用路径。他们通常用浏览器自动化工具来通过用户界面自动化测试。现在不再有手工脚本测试了，但是他们做了很多2探索测试，包括尝试系统的不同路径并尝试破坏它。

一旦所有的Cucumber场景通过了，就将变更部署到发布环境，然后在当天某个时候再推送到3生产环境。

一般来说，uSwitch团队不会跟踪许多技术上的项目指标。相反，他们只看延迟时间和吞吐4量。他们更关注系统的业务性能和功能增加的价值。因此，他们在生产环境里监控转化率(conversion rate)和功能使用率等用户体验指标。

我访问uSwitch的时候，他们的团队正在抛弃估算。估算在商业用户不信任开发团队或他们要投入大量工作的时候有用。但是现在，这些情况都不适用于uSwitch。商业用户对开发有了较多的认识，对开发人员也比以前信任得多。他们一般还是做小型的增量工作。估算一件工作需要花费多少时间就变得没必要了。

12.4 结果

在uSwitch，一个功能的平均周转时间（从接受开发到上线）现在是4天。当我采访他们的时候，他们不记得过去6个月里有任何严重的生产问题，往返的情况也很少发生，只不过几个月发生一次。2009年，在Agile Testing UK用户组Hemal Kuntawala的演讲中，①uSwtich的一个开发经理说：“质量大幅提高了，转化率也增长了。”

注释：①请参考http://skillsmatter.com/podcast/agile-testing/how-we-build-quality-software-at-uswitch.com 和 http://gojko.net/2009/10/29/upgrading-agile-development-at-uswitch-com-fromconcept-to-production-in-four-days。

现在整个开发流程由功能所预期的商业价值驱动着。没有大的计划和大的发布，他们构建小的增量，经常发布，并监控增量是否带来商业价值。因为他们的商业模型依赖于即时网页转化率，所以他们可以很容易地实现这种评估。

从Mark Durrand和Damon Morgan在Spa2010②上的演讲中，你可以看到一些流程如何演化的有趣度量。

注释：②www.slideshare.net/markdurrand/spa2010-uswitch

12.5 重要的经验教训

对我来说，这个故事最重要的一个方面是uSwitch决定专注于提高质量，而不是尝试实施任何特定的流程（具体参见4.1.2节）。他们没有采用一种带来巨大改变的方式，而是时常寻找最重要的需要改善的事情，并立即着手改进。等到对所产生的变化感到满意的时候，他们会继续观察流程，并移到下一个问题。

团队意识到测试是瓶颈，QTP成本太高，对开发人员来说太笨重，这促使他们通过功能测试自动化引入了实例化需求说明，我在4.1.3节中曾提到这种方式。他们最初使用Cucumber是将其作为自动化功能测试的方式，但是后来发现从中获益良多，因为这让他们在自动化测试的同时，还能将它们保持为可读的形式。这使得他们有关需求说明流程的思路完全改变了。

故事的另一个大的收获是改变，虽然改变一开始是通过一个工具驱动的，并且大多是文化上的。uSwitch移除了测试人员和开发人员之间的界限，去掉了测试人员角色，确保所有的团队成员都能理解质量问题是大家的问题。他们开始专注于交付商业价值，而不是完成技术上的任务，这帮助他们提高了商业用户在开发流程中的参与度。没有这些商业用户的紧密合作，就不可能在这么短的时间里决定构建什么，并达成一致意见，然后实现并完成验证。

商业用户更多地参与进来意味着他们开始非常理解并信任开发团队，开发人员也能学习更多的领域知识。正式的需求说明工作坊是构建这种知识的重要一步。沟通改善了，开发人员学习了大量的领域知识，正式工作坊就不再必需了。这是团队知识构建后流程如何优化的一个好例子。

在我的脑海中，uSwitch采取的最具争议性的步骤是决定在功能实现后禁用不太重要的测试。我见过也听过其他团队的别的想法，而他们是唯一一个没有频繁运行活文档系统中的所有测试的团队。可执行需求说明对他们来说绝对是可执行的：可以运行它们，但不是非运行不可。团队发现在开发一个功能的时候运行这些测试有很大的价值，但是随着测试个数的增多，较慢的反馈带来的长期成本比对抗低风险领域的功能回归要高。这可能是因为他们有其他方式来对抗生产环境，特别是持续用户体验监控系统中的问题。

第13章 RainStor

RainStor是一家英国公司，开发大容量的数据归档和管理系统。RainStor的案例分析非常有趣，因为他们处理的技术领域比较复杂，数据量很大，对性能的要求也很高，需要先进的压缩和数据管理算法。

这家公司的员工人数少于30人，而且大约有一半的人在做研发，因此他们在开发与支持他们的软件时必须非常高效。所有开发人员和测试人员都在同一个Scrum团队里工作，不过他们正在考虑分成两个团队。

他们采用实例化需求说明的过程是非常自然的，没有大型的计划或者专业术语，而且主要由测试人员推动。Adam Knight是RainStor的高级测试人员，同时也是支持团队的负责人，当我采访他时，他说：“公司的其他人都不知道什么是验收测试驱动开发。”尽管他们的过程几乎包含实例化需求说明的所有关键要素，但他们认为那只是他们土生土长的软件开发方法。他们使用实例来描述需求，将它们自动化成可执行的需求说明，并且频繁地对其进行验证，以建立起一个活文档系统。他们实现的变革可以让他们的开发团队规模在今后3年中扩大两倍，与此同时还提高了效率。

13.1 改变流程

3年前，一位新的CEO决定采用Scrum，同时扩张了只有4名开发人员的团队，招募了两名测试人员和一名测试经理。Knight说，尽管他们采用了迭代和每日站立会议，但他们的过程实际上还是小型瀑布模型。他解释道：

“在**sprint**开始时我们会有一个很大的需求文档。它既是需求文档，也是技术规范，包含了太多的技术细节。在迭代开始时未对文档做任何修改。随着迭代的推进，开发有时会超出文档的约束，而测试仍然遵守文档的原有内容。在开发过程中文档没有随着开发的改变而及时维护，所以最终我们的测试用例与实际的实现会有所不同。**”**

除了开发和测试的协调问题，他们还遇到了测试执行方式的问题。尽管他们有一些自动化测试，但大部分验证都是测试人员手工进行的。随着产品的发展，手动测试显然无法满足需求。尽管他们增加了人手去执行手动检查，但他们的软件要处理大量的数据，手工检查成千上万的返回结果并不可行。

在2007年年末，Knight担任了该项目的测试经理。他想让测试更加有效，并可以支持开发工作，避免在产品开发过程中进行手动测试。他们实现了一个简单的自动化测试工具，可以让他们更早地在过程中展开测试。在他们开发相关功能的同时，他们还可以制定出测试。这帮助他们保持了开发和测试的一致性。

功能测试自动化给予了他们即时的价值，因为迭代结束时他们不再有成堆的测试任务。同时这还能让开发人员更快地获取到反馈(诸如某项工作是否完成)，避免由于测试拖延到下一轮迭代而造成的流程中断。

一旦团队保持了测试与开发的一致性，他们就会开始注意到范围蠕变的问题，并可以了解到自己何时完成某项工作。开发开始后，他们经常不得不重写需求。之前迭代中的有些工作往往会以名为“清理……”的用户故事的形式再次出现。在2008年的夏天，Knight邀请了David Evans作为顾问，帮助他们理解如何改进。

结果，他们开始使用用户故事来描述范围，而不是预先使用大的、详细的技术需求。这让他们开始从业务角度去考虑验收条件并据此衍生出测试，而不再是以被告知需要实现什么功能的方式来获知需求。Knight说这让他们可以更好地理解范围，并让他们对自己何时开发完某个功能有一个清晰的认识。

他们开始把故事分割成更小的可交付项，这赋予了他们更高的可见性，让他们可以了解一轮迭代实际可以交付什么东西。这帮助团队更好地管理了商业用户的期望。

然后团队开始使用实例来描述满足的条件，甚至是诸如性能之类的需求。Knight解释道：

“我们为性能测量使用了定义良好的验收条件。例如，在多少个**CPU**的情况下，系统必须在**10**分钟内导入一定数量的记录。可以是开发人员访问专用的测试硬件，也可以是由测试人员执行测试并提供反馈。**”**

关注用户故事可以让商业用户更好地参与对即将开展的工作设定期望，而用实例描述这些期望可以让团队客观地衡量他们是否达成了目标。

随着客户基础的成长，他们有更多客户专用的场景要实现。在2008年年末，团队决定把客户作为最终的利益相关者，并让他们参与到需求说明的制定过程中。Knight补充道：

“通常客户知道他们想要在产品中放入什么功能。他们会给我们需求，我们会一起舍作获取真实的数据集与预期目标。我们会把它放入到测试工具中并用其驱动开发。**”**

把带样本数据的客户专用的场景放入系统作为验收测试可以确保团队达成他们的目标。同时这还意味着团队无需浪费时间去想一套单独的验收条件集，并可以防止任何由潜在误解造成的无谓返工。

当实际客户可以参与实际需求的制定时，这样的过程效果最佳。RainStor主要与经销伙伴一起工作，经销商有时会提出一个没有具体业

务用例的需求，Knight称：“这样的需求是最难的。”在这种情况下，他们会向经销商询问实例，有时会组织一些工作坊，与客户一起在开发完的原型上检查高层次的实例。今后他们会用那些高层次的实例去驱动范围。研究纸质原型也可以帮助他们先看到系统的输出，改善由外到里的设计。

13.2 当前流程

目前，RainStor研发团队的迭代周期是5周。他们的Sprint始于周二，在启动会议上会对下一轮迭代将要开发的故事进行简单的检查。他们会利用当天的剩余时间详细说明那些故事。开发人员、测试人员、文档工程师以及产品经理会进行协作，充实需求的细节，并为每个故事制定一些基本的验收条件。测试人员会根据会议记录写下验收条件，并发布给整个团队去看。

故事的满意条件发布后，开发与测试会并行进行。开发人员致力于让现有的带实例的需求说5明通过测试，同时测试人员致力于创建更多详细的测试用例。对于某些故事，起初可能没有自动化任何实例。这种情况下，开发人员会先交付基本的功能，而测试人员则自动化更为简单的实例。然后测试人员继续开发更进一步的测试，同时开发人员交付出确保可以通过那些测试的功能。

当功能实现后，测试人员会执行探索性测试并针对系统的新版本运行自动化测试。开发人员完整实现故事描述的功能后，测试人员会确保所有测试都通过，然后将它们集成到持续验证系统中。

3处最能带给人顿悟的地方

我让**Adam Knight**挑选出**3**处有关实例化需求说明最重要的经验。下面是他的描述。

开发自动化测试工具时，如果恰当地对它们进行设置，揭示出背后的意图，那么它们就可以转变成测试文档。元数据可以让测试的可读性更好。我们会生成**HTML**格式的报表，列出执行过的测试以及它们的意图。调查任何回归失败的原因变得更加容易。你可以更容易地解决冲突，因为你不用回头去看文档就能了解它的意图。

验收条件以及带实例的需求说明是故事创建过程的一部分，它们就是需求。你可以用轻量级的故事开始。一旦拥有了测试，那么测试通过就意味着需求已经满足。你无需参考任何其他东西就可以找到需求。如果将来有需求会引起冲突，我们马上就能发现哪些修改影响了现有功能。这让我们可以持续地对需求进行维护。我们总是可以知道产品对所实现的需求支持得如何。如果测试开始失败，我们马上就可

以知道哪个需求没能符舍预期。

测试以及测试结果是产品的一部分。你应该把它们与产品一起保存在版本控制系统中。我们会测试不同的分支和版本，并且需要执行那些适舍于分支的测试。

有些测试会涉及非常大的数据集，或者需要检查性能，因此他们把持续验证分成了3个阶段：常规构建、通宵构建以及周末构建。常规构建少于1个小时。比较慢的检查会通宵运行。检查非常大的数据集，通常是客户场景，只在周末运行。由于这种反馈很慢，他们只在功能稳定时才会把测试加入到通宵构建或周末构建包中。当开发人员发布部分功能时，他们会尽可能在自己的机器上运行测试。测试人员会运行那些需要专门硬件的测试，并给开发人员提供反馈。

在迭代的最后一周，他们会关闭所有没有解决的问题。他们的团队会确保所有测试都在适当的自动化包中运行。他们会紧密跟踪利益相关者，修正那些已知的问题。在迭代的最后一个周一，他们会运行最后的回归测试并举行回顾会议。

因为RainStor是一家相对较小的公司，所以他们的产品工程副总裁需要负责分析及其他诸多事项。他很难常常参加所有的需求说明工作坊，因此有时候测试人员会介入，帮助他做一些分析任务。测试人员负责搜集问题列表，并在编写带实例的需求说明前获得澄清。

13.3 重要的经验教训

尽管开发团队的规模在过去3年中增加了两倍多，RainStor的团队仍然相对较小。开发产品、支持已有客户以及拓展客户基础都由同一拨人负责。那么少的人做那么多的事，他们行事必须显著有效。下面列出了他们取得的成就及其方法。

实施可执行的需求说明消除了维护两套文档的需要。它有助于保持测试与开发的一致性，并可以消除很多无谓的返工。

切换到用户故事帮助他们让商业用户更好地参与进来。

从商业目标中获取带高层实例的范围，确保他们构建出了正确的产品，并且没有浪费时间去开发不必要的功能。

让客户协作进行需求说明帮助他们让过程更加有效，因为他们从一开始就获取到了验收条件，这可以确保他们达成目标。

过去3年中他们在逐步改善，但他们并没有任何大型的计划，也没有强制实施任何特定的过程。同其他许多故事一样，他们总是在寻找下一处可以改善的地方，在社区中搜寻好的想法帮助他们进行改善，然后想办法在他们特殊的背景里进行实施。这让他们实施了几个不同寻常的

实践，比如只在系统稳定时才执行手动测试，并使用自制工具从元数据中建立起活文档。

特定的背景使他们无法使用任何更为流行的工具来取得同样的效果，因此他们开发了一个工具帮助他们更有效地进行工作。他们从过程入手，并开发了一个工具为其提供支持。

对我而言，这个案例的主要经验是在进行改善时要把重点放在重要的原则上，仅把社区上流行的实践作为灵感来运用。

第14章 爱荷华州助学贷款公司

爱荷华州助学贷款公司是一家财务公司，他们将实例化需求说明的理念发挥到了极致。他们这个案例十分有意思，因为他们的活文档系统为业务带来了竞争优势。这使得他们高效地进行了主要业务模式的变更。

爱荷华州助学贷款公司开发团队建立并维护一个复杂系统，从接受贷款申请的公共网站，到处理保险核保(underwriting)及始发(Origination)的后台系统。此外，项目的复杂性主要来自于业务的数据驱动特性。

我采访了Tim Andersen、Suzanne Kidwell、Cindy Bartz以及Justin Davis，他们在公司改善软件流程的过程中参与了多个项目。从改变小项目到重写整个保险核保平台，追溯他们如何启动这些实践非常有意思。

14.1 改变流程

2004年，爱荷华州助学贷款公司开发团队依照书上的方式实施了极限编程来改善软件质量。第一个项目上线后，他们准备用类似于以往的方式来处理缺陷。接下来的12个月里，新系统只出现了6个缺陷。这向管理层证明了敏捷开发（尤其是测试先行）是一个很好的理念，并且它显著地改善了质量。

然而，测试是一项技术活。团队使用了HTTPUnit（用于对网站做单元测试的框架）。开发人员将用例转成了几乎无人能看懂的HTTPUnit测试。当系统上线时，他们发现缺少文档。之后他们聘请了咨询师J.B.Rainsberger来帮助他们找出其中的问题，并帮助他们使用一些工具与做法来加以改善。FitNesse就是他引入的一个工具。

2006年7月到8月间，团队以使用FitNesse捕获需求说明的方式宣告完成了第一个项目。这个项目使得团队学到了如何使用工具，同时还引导他们重新思考如何编写可执行的需求说明。业务分析师有技术背景，

结果他们与开发人员编写的需求说明太过于技术化。这带来的后果是商业用户无法理解。Justin Davis对此问题的解释是：

“我做为业务分析师能够看到并阅读它们，我们仍然在编写测试，然而团队的其他业务人员基本无法理解这些测试。**”**

要重写整个保险核保平台并将之前在纸上手工完成的许多工作自动化、还仅仅是大量工作的开始。接下来的项目将需要花费整个团队3年的时间，这个团队包括6个开发人员、2个测试人员、1个业务分析师以及1个商业用户。他们引入了一名咨询师来帮助改善团队与商业用户的沟通。Tim Andersen说：

“David Hussman说我们应该加倍努力使测试写得更清晰明了，这样我们无需解释，业务人员也能够轻易读懂。但是这并非轻而易举的事情。我们需要转变思维，需要更强的业务理解能力，还需要更多地了解系统应当如何运作，并就此进行沟通，而不仅仅是只了解技术需求。**”**

随后，他们开始以用户角色来描述系统，这样可以考虑不同种类的用户是如何与系统交互的。

他们不再使用通用的用户，而开始更多地考虑不同用户使用系统的目的，他们想要从中获得什么以及将会如何使用。这使得业务利益相关者更多地参与进来，并能为团队提供更有意义的信息。Tim Andersen在2010年的code Freeze大会上的演讲里提到了不少他们使用的关于用户角色的好例子①。

注释：

①http://timandersen.net/presentations/Persona_Driven_Development.pdf

14.2 优化流程

由于他们之前的可执行需求说明非常技术化，所以自动化层非常复杂且难以维护。测试将技术化组件描述为大流程的一部分，因此他们必须通过伪造用户流程的某些部分才能完成自动化。Tim Andersen说测试结果也不可靠：

“测试通过了，然而软件却无法使用。测试在撒谎（伪绿色）。例如，一个**18**周岁以下的借款人却可以借款。我们将会写一个测试，如果借款人小于**18**周岁，测试会提示：‘没有共同签署人的情况下你不允许借款。’而如果更改借款人的出生年月，使其大于**18**周岁，测试将提示：‘你可以借款，无需共同签署人。’这样测试就通过了，然而当我们在浏览器里试用这个正在开发的功能时，却无法使用。我们编写了

验证规则，却没有放在正确的地方，无法起到真正的作用。我们的测试代码所测试的是‘虚幻的状态’。”

商业用户不信任可执行需求说明的测试结果，因此也不觉得测试很重要，这也是妨碍他们更多参与其中的另一个障碍。Andersen说：

“双方都很有挫败感。我们质疑他们：‘为什么不评审测试，为什么不重视测试？’而同时业务团队也感觉很沮丧：‘为什么开发人员写的测试通过了，而软件却无法使用？’他们不相信这些测试。”

为了跑通整个产品流程，团队重整了可执行需求说明的自动化层，不再使用虚假状态。需求说明新的自动化方式非常适应描述系统所使用的用户角色。Andersen说：

“我经常使用‘虚幻的状态’这一词，目的是为了使开发人员明白那些不使用正确切入点的测试是无法取得我的信任的。虚幻状态的其他症状是‘**fixture**太笨重’，**fixture**应该没有太多逻辑，应该相当轻巧。使用角色有助于我们找到正确的抽象级别，以便在系统中找到恰当的切入点。未使用角色之前，我们选择的切入点往往不够恰当，结果导致了出现笨重的**fixture**，而且容易出现‘虚幻状态’。”

团队依照角色的活动来组织自动化。每个角色都以fixture的形式在自动化层实现，并使用HTTP请求与服务器交互，其本质上就是不打开浏览器却达到与使用浏览器一样的效果。这不但极度简化了自动化层，而且还使得测试结果更加可靠。使用这种方式后有些测试开始出现错误，团队因此发现了原先并未注意到的缺陷。大约在2007年5月份，测试结果变得可靠多了，并且自动化层也更加容易维护。Andersen补充说：

“将测试代码更改为依赖应用程序设置‘贷款’的状态，这样一来就暴露了缺陷，我们就有机会做修复，并且‘伪绿色’症状也销声匿迹了。而且还极度降低了测试的维护成本。”

只要可执行的需求说明用于描述业务功能的措辞是商业用户能够理解的，那么自动化层就会变得简单很多：与业务领域代码相关联。它还使得需求说明更有相关性，因为测试的流程完整并且不再产生“假阳性”的误导。

随着测试个数的增加，反馈开始变慢。许多慢的偏技术化的测试是通过浏览器进行的。Andersen说依照角色来看待整个系统有助于减少此类问题：

“我们使用**FitNesse**作为配置**WatiJ(**一个**UI**自动化库**)**的工具。未使用角色之前，我们使用浏览器进行测试是不得已而为之的，因为‘我们必须这么测试才能确保功能确实没问题’。此类浏览器测试层出不穷。”

团队使用“角色”重写了浏览器测试，这显著地改善了反馈时间。新的自动化层直接发出HTPP请求，无需每次都打开浏览器。他们还研究了如何取代SQL Server，在内存数据库中运行测试，但是他们最终决定在真实的SQL数据库中使用索引来提高性能。团队将持续验证流程分成几个模块，以便更好地观察是什么使得测试变慢。

当进行需求变更时，团队开始考虑如何与原有的需求说明集成，而并非每次都创建新的需求说明。这减少了测试的个数并有助于避免不必要的配置工作。Andersen解释道：

“我们开始考虑场景。一个新的功能可能不是独立的，它有可能是对一组场景的变更。对此，我们不是为每个需求编写新的测试，而是以目前系统所具有的上下文来考虑问题，并且考虑‘需要改变哪些测试’而不是‘需要编写哪些新的测试’。这样就不会增加构建的时间。”

这还帮助他们开始重组需求说明以便减少测试的个数。他们会寻找较小的部分的需求说明，并将其合并成较大的需求说明。他们还会将大型的需求说明分解成几个较小的、更专注的需求说明。“基本上必须像重构代码一样地重构测试与测试代码。”Andersen说道。

爱荷华州助学贷款公司是实例化需求说明的早期使用者，因此他们不得不使用还未成熟的工具，这些工具多次妨碍了协作。因为团队使用的是开源工具，所以他们可以依照自己的开发流程来更改这些工具。

在他们将可执行需求说明放入版本控制系统之后，业务分析师如果不使用开发工具就无法直接对其进行更改。开发人员为FitNesse编写了一个可以集成版本控制系统的插件，这样业务分析师仍然可以在wiki上更改需求说明。

测试的个数变得越来越多，团队在功能回归测试方面出现了问题。本应该被现有的测试捕获的缺陷却成了漏网之鱼，因为相关的测试被禁用了。有些测试是由于开发人员不确定如何与新功能衔接而被禁用的，而有些是因为团队在等待业务利益相关者的决定而被临时禁用的。之后人们就忘记了重新启用这些测试，或者忘记跟踪讨论。爱荷华州助学贷款公司的开发人员为禁用的测试编写了一个自动检测（详见10.3.2节），这样可以在每个迭代结束时提醒他们还有哪些事情需要后续的行动。

他们使用JIRA管理需求，使用FitNesse管理可执行的需求说明，因此重新安排FitNesse页面导致了JIRA里的链接不再起作用。他们对FitNesse做了扩展，使其支持关键字，然后在JIRA里使用关键字链接到可执行需求说明的页面。在另一个项目里，他们采用了一种不同的方式并创建了一个业务框架。该业务框架是一组FitNesse的页面，其设计初衷是作为稳定的文档切入点，这样就有了连到测试的内部链接。这是好

的活文档系统的开端。Justin Davis解释道：

“业务框架的一个目的是创建一个**FitNesse**的前端，这样不仅业务团队能够使用，而且开发人员也能理解当前系统的内容。事实上，它提供了系统如何运作的映射图。因此，只要有个上下文，比如知道系统如何运作，就能够从这个框架里找到你想要的。有了系统工作流，你就可以选择想要查看的步骤的测试与需求。”

引入业务框架并且确保可执行的需求说明保持相关性并确实被频繁地验证，这样使得他们可以创建一个有用的活文档系统。他们有一个相关的信息源，任何人都可以使用它来了解系统的功能。

14.3 活文档作为竞争优势

有了如此完美的活文档系统，他们就能够非常高效地处理重大变更。在项目结束前3个月，公司的业务模型突然需要变动。过去通常是通过出售债券来资助贷款。由于2008年的信用危机，债券出售失败了。由于公司的业务受技术驱动，因此业务模型的变动也必须在软件中相应地体现。Andersen说活文档系统帮助了他们，让他们了解怎样才能为这个业务变更提供支持：

“通常情况下，我们使用债券收益资助私人助学贷款。然而我们改变了业务模型，使得系统中所有的资金都可以配置。这样，贷款出借人可以作为资金提供者，我们就可以继续为学生提供助学贷款。这绝对是一次对系统核心的大翻修。有这个新需求之前，系统甚至都没有贷款出借人的概念，因为我们都默认爱荷华州助学贷款公司就是贷款出借人。

原有的验收测试都能够派上用场，只需说‘这是我们对于资金的新的需求’，做适当的改变后就可立刻发挥作用。我们针对所有的测试讨论了影响并提供资金，这样他们仍然可以运作。对于资金不足的情况，或者资金充足但某个学校或某个出借人无法使用等情况，我们做过有趣的讨论，因此我们对于这些需求拟定了一些边缘情况，但是这只是为了新的资金模型可配置以及更具弹性。”

一旦他们理解了新的业务模型对软件带来的影响，他们就能够高效地实施方案。据Andersen所述，如果没有活文档系统就无法快速实施此类变更：

“因为我们有很好的验收测试，所以我们能够在一个月之内实施解决方案。如果没有这些测试，任何系统碰到类似的情况都会导致开发中断，并且需要重写系统。”

这就是在活文档系统上的投资回报。它在分析、实施以及测试业务

模型变更所带来的影响等方面提供了支持，同时可以让他们快速验证系统其余部分是否受到了影响。

14.4 重要的经验教训

他们一开始关注工具，之后很快意识到工具无法帮助商业用户在流程中发挥作用。所以他们开始以用户的角度来使用需求说明。这使得他们能更好地与商业用户沟通，并且还降低了测试的维护成本。当工具妨碍了高效的协作，他们就动手更改工具。这也是使用开源工具的另一个理由。

爱荷华州助学贷款公司实施实例化需求说明不是出于改善质量或自动化测试的目的，而是出于建立相关文档系统的需要，以便使系统更加高效，并且吸引商业用户参与其中。他们大手笔投资以建立好的活文档系统，而结果获得了丰厚的回报。活文档系统帮助了他们实施业务转型，体现出了相当的威力。

第15章 Sabre Airline Solutions

Sabre Airline Solutions提供软件和服务来帮助航空公司制订计划、运营以及销售他们的产品，他们较早地引入了极限编程与实例化需求说明。这是一个十分有趣的案例，因为他们在一个大型项目和庞大的分布团队上应用了实例化需求说明。

这个项目是Sabre AirCentre Movement Manger，这个软件系统监控航班运营，并在发现问题时通知相应的团队，帮助他们调整日程，将对客户和航班的影响最小化。据Sabre的敏捷教练Wes Williams所述，由于领域的复杂度和质量问题，之前两个构建类似系统的项目都失败了。实例化需求说明帮助他们成功地完成了这个项目。

15.1 改变流程

实施极限编程不久后，因为领域的复杂度，Sabre的团队在寻找一种能够在极限编程实现后即对验收测试进行说明并自动化的合作方式。Williams说他们最初尝试的是偏技术化的单元测试工具。不过这种方式并没有改善合作，并且还不能重用，所以他们放弃了。

他们开始寻找一种驱动合作的工具。2003年，Williams发现了FIT，也就是第一个广泛使用的自动化可执行需求说明的工具。他的团队开始用FIT实施验收测试，但是他们太过于专注工具而非实践。Williams说这

并没有给他们带来预期的合作上的改善：

“我们喜欢的想法是客户可以定义测试并驱动应用所带来的价值。事实上，从未有一个客户用**HTML**编写过**FIT**测试。大多情况下测试是由开发人员编写的。让客户做到这点并不是那么容易。测试人员使用的是**QTP**。这无法带来协作，因为开发人员不会运行**QTP**测试也不会参与编写**QTP**测试。”

当时编写实例化需求说明的只有开发人员，而他们也知道这无法给他们带来预期的好处。为了改善沟通和协作，每个人都要参与其中。只凭一小部分人是无法独立做到的。

有个产品开发高级副总裁受开发团队的影响，请来了ObjectMentor的顾问来给大家做培训。他们使更多的人知道了实例化需求说明的目的以及可以从中获得的好处。虽然他们没有让大家立刻开始行动，但培训使更多的人对这个实践产生了极大的兴趣。Williams说：

“不是所有人都能接受。但是，仍然有一群核心的人相信这个实践，他们收获很大。而不接受的人依然一如既往地反对。”

核心的这群人从一个相对简单的Web项目开始，它是用于聚合多个软件构建信息的内部系统。为了尝试实践并熟悉工具(这些2004年的工具，在今天看来是相当的落后)，团队选择了FitNesse来协作制定可执行需求说明。基本上，他们编写可执行需求说明是与开发并行进行的，甚至在开发前进行。项目的商业利益相关者是公司内部的一个经理，他参与审查了测试。团队一开始将自动化层看作二等测试代码，不太关心代码是否整洁，结果导致了不少维护问题。而且在他们的测试需求说明中有着大量的重复代码。Williams说：

“我们学习到应该保持测试代码尽可能简单，不要出现重复。测试（自动化层）也是代码，和其他代码一样。”

开发人员对自动化层或可执行需求说明的可维护性不大关心，因为他们觉得这些只与测试相关。他们在项目末期意识到这种做法给项目带来了巨大的维护问题。与爱荷华州助学贷款公司的团队类似，第一个项目让Sabre Airline团队学习了如何使用工具，并看到了他们自动化可执行需求说明的方式的效果与局限性。这样他们也知道了如何在下一个项目中做改善。

小型团队更好地理解了工具的局限性，并且意识到为何要在编写可维护的实例化需求说明上投入更多的精力，之后，他们开始在大型的并且具有一定风险的项目上应用这个流程。该项目用Java重写C++写的遗留系统，需要多次交付。这是一个数据驱动的项目，并且需要支持全球分布。最后，30个人一共花了两年的时间交付了整个项目，并且是由分布在两个大洲上的3个团队共同完成的。

由于风险，他们想要大幅提高测试的覆盖率与测试频率。于是他们用上了之前在小型项目中实施的做法。Williams说：

“如此大型的应用如果用手工测试需要几个月。我们想要防止缺陷，并且不想花上几个月的测试时间。于是我们采用了持续测试。你不可能每天在如此大型的应用上手工做可行性测试。”

他们内部有了使用FitNesse的经验，因此参与了之前项目的人开始编写自动化功能测试。他们邀请了商业用户参与完成测试，并期望这么做可以达成预期目标。

15.2 改善协作

他们分成了3个团队。第一个团队负责核心功能，第二个负责用户界面，第三个负责和外部系统集成。交付第一个版本的用户界面花了4个月左右。商业用户看到界面后，核心团队发现他们的软件遗漏了许多客户预期。Williams解释道：

“客户看到的界面与他们想象的完全不一样。当我们开始编写用户界面验收测试的时候，用户界面的测试比领域的多很多，所以不得不修改领域代码。但是客户认为这部分已经完成了。他们有自己的**FitNesse**测试，运行之后顺利通过。人们认为后台会处理界面所表现的所有东西。有时对于前端来说可用的查询或数据检索在后台却无法支持。”

他们意识到了问题出在团队之间的工作分配上。客户具有先天优势可以周全地考虑系统细节，但前提是让他们“看到”内容。如果团队不能交付任何用户界面，客户也就无法很好地参与到团队定义需求说明的工作中。

项目开始6个月后，项目组决定重新组织工作，好让团队交付端到端的功能。这使得商业用户能和所有团队合作。Williams补充道：

“我们分了功能组之后，用户故事和应用核心已经非常成熟了，不会有故事爆炸性增长。出现意外的几率小多了。”

当每个团队开始端到端交付整个功能的时候，商业用户与团队协作定义验收标准以及参与到举例说明中就简单多了。

重组工作后，团队发现他们需要得到已实现故事的更快反馈，所以他们将迭代长度减为一个星期。虽然在实现前就编写了验收测试，但是他们依然将其视为测试而非需求说明。测试人员负责编写验收测试，但是他们跟不上这么短的迭代。为了去除瓶颈，之前项目中使用FitNesse的小组建议开发人员应该帮助编写验收测试。Williams说测试人员一开始很不愿意接受：

“一开始让开发人员答案编写测试颇费周折，因为测试人员认为他们测试做得更好。我想这是因为他们考虑问题的角度完全不同。事实上，我发现自从开发人员和测试人员一起讨论测试以后，编写出来的测试会比他们某一方独自编写时好得多。”

Williams意识到这需要文化上的改变。作为教练，他尽量创造环境让大家一起工作，并让他们把问题暴露出来。当测试人员测试落后的时候，他让开发人员来帮忙。当测试人员抱怨开发人员不知道如何编写测试的时候，他建议团队结对编写测试。

“他们体验后都十分惊讶：‘哇哦，我过去写的完全跟这没法比！’你需要让他们都体验一下。”

Williams惊讶于这之后测试人员和开发人员之间建立起的信任：

“信任十分神奇。他们发现，一起做可以做得更好，并且大家荣辱共担，因而不会有人从中作梗。最终他们拥有了更加协作的环境。”

让大家一起工作不仅仅帮助他们处理了流程中的瓶颈，还得到了更好的需求说明，因为不同的人可以从不同角度考虑问题。协作帮助大家共享知识并与团队里其他人之间逐渐建立起信任，长期来看这使得流程更加高效。

15.3 结果

虽然之前两次尝试重写遗留系统都因为质量问题失败了，但这次项目一开始就是一个很大的客户上线了，而且问题很少。他们只发现了一个严重问题，是与故障切换有关的。Williams说实例化需求说明是成功的“关键部分之一”。

数据驱动项目的关键实践

Wes Williams分享了在数据驱动的环境中编写优秀需求说明最重要的**5**大技巧。

隐藏次要数据。

消除重复。

做增量开发时，找出重复部分，看看过去类似的测试并清理。

像对代码一样对测试进行重构。

隔离自己，不要依赖于数据不受你控制的第三方。在航空领域中，系统最终要和许多宿主系统通讯。他们可能也有测试系统，但是其数据不受你控制。你的测试需要与他们通讯，但是这些是完全隔离的测试。在自动化验收测试的过程中，这部分需要模拟。

15.4 重要的经验教训

开发人员通过推动实例化需求说明的使用来接触测试人员和商业用户，但是他们很快发现在封闭的小组中专注于工具是不会成功的。让大家都参与进来是至关重要的。虽然培训不会让大家都开始行动，但也给了他们一个共同的基准，并且还可以发现真正有兴趣尝试新想法的核心人员。

他们使用较小的，风险较低的项目开始实验工具，并且发现编写和维护需求说明及自动化层的好方式。参与了试验项目的那一小组人在大项目里的更大团队中扮演了催化剂的角色。

当团队开始交付系统组件时，商业用户不能正确地和团队协作做后台组件，这导致了大量的重复工作，并且还缺失了不少预期功能。在他们重组成功能团队后，这个问题就不复存在了。

让测试人员和开发人员协作编写验收测试能产生更好的需求说明，并有助于在两组人员之间建立信任。

实例化需求说明通过提供清晰的开发目标和持续验证，来帮助他们征服复杂的领域。

第16章 ePlan Services

ePlan Services是位于科罗拉多州丹佛的一家401k退休金服务供应商①。他们的业务是由技术驱动的，并紧密依赖于高效的软件交付过程。Lisa Crispin是在那里工作的一名敏捷测试人员，她说他们不仅使用活文档来处理复杂的业务领域，而且还用它帮助软件开发人员和业务操作人员之间的知识传递。

注释：①401k计划是一项基金式的养老保险制度，指的是美国1978年《国内税收法》新增的第401条k项条款的规定。——译者注

该公司的业务模型是给那些小型企业提供服务，由于运营成本低廉，因此他们更具有竞争优势，这其中业务流程自动化是一个关键因素。2003年，他们意识到为了支持业务，他们的软件交付过程必须有所改变。Crispin说：“当时我们的软件都没法发布，因为有太多的质量问题。”

他们需要交付更为廉价的服务，并需要将业务流程自动化，这些现实需求驱使ePlan Services公司走上了一条改善软件开发过程的道路，在此期间他们实现了实例化需求说明的大部分想法。

16.1 改变流程

公司说服了Mike cohn去接管开发团队，他帮助他们实现了Scrum。刚开始的时候，他们在每轮迭代的最后两天执行手工测试。团队的所有成员，包括测试人员、开发人员以及数据库管理员，都会去执行手工测试脚本。这意味着五分之一的迭代时间都花费在了测试上。为此，他们决定实现测试自动化。Crispin说，他们首先要纠正的是单元测试：

“以前测试人员找到的缺陷大多是单元级的。我们在这上面投入了所有时间，因而没有时间去处理其他事情。”

当开发人员习惯于单元测试时，测试人员开始进行功能测试自动化。没有开发人员的帮助，测试人员只能通过用户界面去自动化测试。8个月后，他们拥有了数以百计的单元测试，而且还自动化了足够多的功能性冒烟测试，这使他们不必再执行手动回归测试以检查单元级缺陷。Crispin说这让他们可以从更宽广的视角来看待产品：

“我们很快就发现，一旦开发人员掌握了测试驱动开发，我们就不会再有那些单元级的缺陷了。我们有更多的时间去做探索性测试。报告的缺陷也通常是因为开发人员没有理解需求，产品中发现的缺陷往往是由于我们对某些东西没有完全理解。”

就像其他很多案例一样，没有高效的测试自动化，团队就没有时间去处理其他任何事情。一旦技术上的单元级缺陷不再引发问题，那么他们就可以看到其他问题了。

尽管他们有一些自动化的功能性测试，但这并不足以防止问题发生。由于这些测试执行缓慢，只能通宵运行并且只检查了测试通过路径的场景。团队开始寻求其他方法来自动化功能测试，以便更快速地运行更多检查。他们找到了FitNesse，但是这种自动化需要开发人员的帮助。Crispin说让开发人员参与进来是一个挑战：

“程序员习惯于因编写产品代码而得到奖励。按照**Mike Cohn**的建议，我拿起一个故事卡，走到正在开发这个故事的开发人员那里，并问他我们是否可以结对编写**FitNesse**测试。下一个**sprint**我会拿一个不同的故事选一个不同的开发人员。我们马上就找到了一个缺陷，实际上是他没有真正理解需求。因此开发人员马上看到了那么做的价值。”

协作编写测试让测试人员和开发人员可以一起讨论需求，并可以帮助他们编写出更好的测试。Crispin说这消除了大部分较大的问题：

“在实施敏捷后的一年里，我们觉得很满意，因为产品里没有出现很严重的缺陷。”

同时他们也意识到了协作的重要性。Crispin说：

“这种方法的最大好处在于它让我们一起进行讨论，这样我们对需求就会有共同的理解。这比测试自动化本身更为重要。我们获得了协作的好处，而产品负责人对验收测试驱动开发也喜闻乐见”。

高效的功能测试自动化需要测试人员的参与，这会使开发人员与测试人员之间的协作更加紧密。同时它还给予团队明显的好处，帮助团队为进一步改善而建立起一个成功的业务案例。这还可以鼓舞团队更加完善他们的过程，防止缺陷进入系统而不是等它们进了系统后再用自动化测试去找出来。

更进一步，他们开始把验收测试作为需求说明，并协作定义验收测试。Crispin与他们的产品负责人一起预先准备实例。他们取得了一些早期的成功，但仍然把实例看作是功能测试。一致性测试是系统最复杂的一部分，当他们使用这种方法着手进行一致性测试的自动化时，他们过度地说明了测试，导致不得不对自己想要达成的目标考虑更多。Crispin解释道：

“产品负责人和我一起坐了下来，为测试算法编写了所有的**FitNesse**测试。由于排列组合的情况太多，我们事先花了好几个**sprint**的时间编写了许多非常复杂的测试。当开发人员开始进行编码，看到测试时，他们感到困惑。因为这让他们看不到系统的全貌。**”**

他们意识到开发人员无法事先处理过多的信息。经过几次试验后，团队决定事先只编写高层次的测试，以便让开发人员能掌握全局概况。当一名开发人员选了一个故事，他会与一名测试人员进行结对，编写主要逻辑场景的测试并进行自动化。然后测试人员会通过增加更多实例去扩展需求说明。测试人员利用自动化框架来探索系统。如果他们发现某种情况会让测试失败，那么他们会回去找开发人员将其修复。这种方法改变了他们将验收测试当成需求说明的方式，Crispin说：

“起初我们有一个模糊的想法，认为自己可以事先编写好验收测试，然后把它们当成需求。随着时间的推移，我们不知道自己事先需要进行多详尽的测试，多少测试才够用。我是一名测试人员，或许我可以不断地测试某个功能，并不停地考虑围绕该功能还需要做什么测试。但我们只有两周时间，因此我们必须解决如何把风险分析内部化，并且能够说：这些就是我们真正需要的测试；这些就是用户故事真正重要的部分，它们必须正常运转。**”**

当他们的思维从自动化测试转向到自动化需求说明时，很明显他们试图详细说明并进行自动化的物件，其主要作用是充当一个交流工具，而不是回归检查的工具。他们对需求说明进行了简化和提炼，以确保开发人员在需要时可以及时获得足够的需求说明。

优秀的测试设计

Lisa Crispin是一位知名的敏捷测试人员，也是**Agile Testing**一书的舍著者。我问她怎么做出优秀的验收测试设计。她的回答是下面这样的。

优秀的测试设计关键在于要长久。大家开始进行测试，继而编写了大量测试。突然之间，维护它们的投入超过了测试本身的价值，这就不是好的测试设计。

每个测试必须清楚它的本质是什么。

一旦发现重复，就必须将它抽取出来。

测试的设计应该由程序员或者拥有较强代码设计能力的人员协助完成。有了模板，放入细节就很容易了。

16.2 活文档

如果团队把实例更多地看作是需求说明而不是测试，就会意识到它们是非常强大的文档。Crispin说当他们调查问题时，拥有一个活文档系统帮助他们节约了许多时间。

“我们接到过一个电话：‘我们有一笔贷款的利息金额不对。我们认为这是一个缺陷。’我可以查看**FitNesse**测试并输入那些数据。也许是需求不对，但目前代码就是那么工作的。这可以节约很多时间。”

有一次，ePlan的某个经理，同时也是一名高级开发人员，决定回印度，他会有几个月的时间无法与团队一起工作。Crispin说他们开始应用实例化需求说明来提取他独有的系统知识：

“每当出现奇怪的问题，他总是知道怎么进行修正。因此我们必须了解他掌握的系统遗留部分的知识。我们决定每个**sprint**都要有一个人花些时间去仔细检查一部分业务流程，并记录下来。”

这还帮助他们开始记录系统的其他部分。尽管他们会为任何正在开发的东西编写测试，但仍然有部分遗留系统没有进行测试自动化，有时候这会引起一些问题。为这些地方建立起一个自动化的活文档，帮助他们发现了业务流程中的不一致。Crispin解释道：

“那时我已经在公司待了**4**年，但我一直不了解现金会计是怎么工作的。我了解到在自动化程序外，我们有**5**个不同的银行账户。这些账户里的钱通过电子邮件和电话转来转去，但是现金数额必须平衡。一旦发生不平衡，会计师需要一种方法去做调查。在会计师向我们解释了这个过程后，我们在**wiki**上记录下来以备后查。然后我们就可以撰写报告，提供金钱转入转出系统的有用信息。现在，当现金失衡时，会计师可以利用报告找出问题。”

活文档系统的建立帮助开发团队分享了知识，让他们学习了业务流程，同时还让业务人员清楚地了解到系统实际在做什么事情。把东西写下来可以暴露出不一致以及有分歧的地方。在本案例中，它让大家努力地从业务角度去考虑系统实际上在做些什么。

16.3 当前的流程

所有这些变化都是在不久前完成的，而且有了活文档系统，团队就有了一个相对稳定的过程。目前，团队由4名程序员、2名测试人员、1名Scrum大师、2名系统管理员、1名数据库管理员以及一名经理组成。他们的迭代周期是2周。每个sprint开始前两天，团队会与产品负责人及项目干系人会面。产品负责人会介绍下一个sprint计划完成的所有故事，他们会在白板上写下高层次的测试。这可以让团队针对计划提供反馈，并可以在实际的sprint计划会议前提出问题。

在这种业务复杂、团队较小的情况下，产品负责人是一个瓶颈。为了让他能够和上游的商业用户一起工作，测试人员需要接手一些分析工作。产品负责人经常会事先创建一个“故事检查清单”，包含故事的目的以及大致的满意条件。对于那些涉及用户界面的故事，他会在故事检查清单上加入用户界面模型。对于那些处理复杂算法的故事，他会加入一张带实例的表单。

最后，产品负责人还会做更多与软件无关的工作，因此他经常没有足够的时间去准备会议。为了解决这个问题，测试人员会征得他的同意，直接与上游的利益相关者接触，并同他们一起制定需求说明。

迭代始于计划会议，会议上他们会再次检查故事，产品负责人会回答任何尚未解决的问题。他们会创建界面模型并使用实例描述需求。测试人员会将那些信息与故事的检查清单合并起来，如果清单上的检查点都具备了，那么他们就会对需求说明进行提炼并放到wiki站点上。

sprint的第4天，两名测试人员会与产品负责人碰头，仔细检查所有的需求说明和测试用例，以此确保他们正确理解了所有事情。这让产品负责人有机会在迭代中对需求说明以及团队将要进行的工作进行一个回顾。

一旦需求说明开始出现在wiki上，开发人员就会开始着手实现故事，完成后他们就会把结果展示给业务人员看。

顿悟之处

我问**Crispin**实例化需求说明让她顿悟的地方是哪里。她的回答如下。

我不控制质量。我的工作是帮助客户理解质量，帮助整个团队定义质量，确保万无一失。

需要开发人员的参与。

这个过程需要耐心。我们必须谨慎前进，不要试图立刻实施所有要素。

像**FitNesse**这种工具的确有助于协作。也许你觉得它是技术上的东

西，会帮助你进行自动化，但它还可以改变团队文化并有助于更好地进行沟通。

实例化需求说明的真正价值在于我们进行交谈了。

16.4 重要的经验教训

由于他们的商业策略，ePlan Services公司对业务流程自动化以及高效的软件交付有着很强的依赖。为了提高质量并加速软件交付，他们必须放弃手动软件测试。最初他们关注于功能测试自动化，但接着他们发现由协作带来的共识可以引导他们开发出更好的软件。

刚开始，他们从测试的角度去考虑协作，过度地说明了测试，这让开发人员很难将那些文档当成开发目标。之后，他们并没有覆盖所有可能的数据组合，而是只说明那些关键实例，这使他们的过程更加高效，还为开发人员及时提供了良好的需求说明。

有了一套针对系统某部分功能的比较全面的可执行需求说明之后，他们就立刻意识到活文档是多么有用，特别是它作为一种记录专家知识的方式。当他们开始记录其他部分的业务功能时，具有一致性的活文档系统就暴露出了他们当前业务流程中的错误与不一致之处。

活文档系统可以让软件交付过程高效得多，还让他们发现了业务过程中的不一致。

第17章 Songkick

Songkick是一家英国的创业公司，它运营着Songkick.com，一个在线音乐的消费者网站。他们是个非常有趣的案例，有两个原因。首先，跟本书中的其他公司不同，他们在创业阶段就开始实施实例化需求说明了，那时尚不涉及处理大型遗留系统的问题。其次，跟本书提到的其他大多数项目也不一样，用户交互是他们的产品中一个最重要的因素，根据对用户如何使用网站的观察，新功能开发很强调用户体验。

系统的复杂度大多源自于许多改善用户体验的微妙之处，以及他们用以为用户提供丰富体验的诸多功能。Songkick实施实例化需求说明是为了专注于交付有效的软件，并且能够帮助开发团队成长。

“作为创业公司，你必须始终都在交付价值。”他们的CTO Phil Cowans如是说。下面是他指出的实例化需求说明最大的好处：

“我们可以更快地获得真正想要构建的东西，因为我们在确定要构建的内容并与客户沟通的整个过程中所使用的语言同测试中使用的语言是一样的。这有助于减少沟通问题。我们不希望出现下面这种情

况，开发人员回头来说：我们做的是对的，只是你要求的东西不对。”

比起更加成熟的公司，对于一个创业公司来说，交付能有效增加实际价值的软件显得更为重要。实例化需求说明的实践帮助Songkick从软件开发投资中获得了更多的价值。

17.1 改变流程

他们的项目开始于两年半以前。Cowans说，第一年过后，团队开始扩大，“一张办公桌已经坐不下了，没法所有人都坐在一起开发”。为了应对越来越复杂的代码库以及不断扩张的团队，他们决定实施测试驱动开发。他说：

“开始**TDD**前，我们发布新代码是基于我们相信之前构建的系统都正常工作。但是不久我们明显感到需要更多的信心，要确信当我们完成某项功能时，它确实按我们设想的工作，并且没有导致回归问题。很明显，如果我们找不到一种方式以避免长期存在的互相妨碍的问题，需求沟通的问题以及出现回归的问题，那么我们的发展速度必定会变慢。”

3个月后，他们发现TDD是比较自然的方式。同时，他们开始研究看板与用户故事的想法，这引导他们开始在业务功能上应用TDD原则，并开始有效地实施实例化需求说明。他们整个团队喜欢实验不同的工作方式，所以他们可以心平气和地开始做尝试。Cowans解释道：

“我们联系了一个在早期项目中使用过看板的人，他现在是正式雇员，但此前是项目的顾问，并非正式雇员。通过他我们相信了使用用户故事是可行的，而且看起来更舍理。将这些加入我们的流程中，这样的决定仅仅是出于下面这样的想法：‘已经有人使用这个技术这么做了，让我们试试看效果怎样。’于是这变成了一件对我们来说很自然的事情。”

团队开始从目标获取范围，从商业价值的角度来驱动用户故事。他们还使用Cucumber创建可执行需求说明。Cowans说随着Cucumber使用得越来越好，流程的焦点从单元测试转移到了业务需求说明上：

“一开始我们混舍使用**Rails**测试框架和**Cucumber**。我们使用**Cucumber**驱动高层用户故事和单元测试来说明具体行为。随着时间的推移，我们越来越多地使用**Cucumber**，并找到了更多地用**Cucumber**来描述事情的方法。”

新的描述方式帮助团队专注于构建真正重要的软件。Cowans说：

“这帮助大家专注于我们做事情的初衷，并看到我们所做的事情的价值。同时还帮助大家避免将时间浪费在构建不需要的东西上。所有

人都从同一个方向来解决问题，这确保了开发团队和公司其他部门都能以同样的方式考虑问题。”

类似ePlan Service的团队，Songkick首先实现（单元）测试驱动开发，然后扩展到商业功能。他们没有真的碰到让他们改变流程的质量问题，但是他们积极改善流程来提高效率。

Cowans说实施实例化需求说明的时候，团队面临的主要挑战是理解要测试什么，如何使可执行需求说明更可靠，以及如何使持续验证更快。

一旦团队习惯于如何使用工具，他们就很容易过分专注于用户界面功能，因为这些很容易考虑。Cowans说：

“我们花了太长时间测试用户界面中很多琐碎的地方，因为这么做很容易。我们没有用足够的时间深入测试边缘情况以及程序的其他路径。”

他们通过一种与用户界面紧密相连的方式来自动化可执行需求说明，所以测试结果并不可靠。Cowans说在某些情况下这迫使他们在开发完成后才做测试，而无法将测试用作需求说明：

“有人修改了网页上的标点就导致测试失败了，这很不好，很令人沮丧，因为这使得一项变更所带来的影响变得难以预估。所以很难实现预先更新整套测试。结果，在某些情况下大家只得先写代码再做测试。

为了解决这些问题，团队开始把测试变得语义性更好，并由自动化层完成领域语言和用户界面概念之间的转译，Cowans解释道：

“你对这个过程不断熟悉，并开始理解依赖于用户界面容易导致长期的问题。开发更专注于领域的步骤的定义（在自动化层）有助于解决这个问题，因它赋予我们更高层的解决问题的方式。”

他们所做的改变实际上是开始提炼需求说明，并寻找用商业语言表达需求说明的方式，而不是用户界面的语言。

根据Cowans的说法，团队用了6个月左右的时间习惯实例化需求说明的流程和工具：

“大概是在最后的**6**～**9**个月，团队才觉得这是我们要做的事情之一。在最后的**9**个月里没人真正质疑我们如何描述工作，它就是这么自然而然地存在着。”

当不得不重写系统的一部分以处理活动订阅的时候，团队才意识到他们的可执行需求说明是多么重要。现有的一套专注于业务的需求说明已被自动化成测试，这使得他们确信在重写订阅功能时没有引入缺陷或减少功能。Cowans说：

“团队所有人都知道此事省了不少时间，而我则觉得测试为我们节

省了**50%**的重构时间。”

对于创业公司来说，节省一个任务50%的时间意味着很多。可执行需求说明有效地防止了系统的回归问题。Cowans说，他们产品环境的问题非常少，因此他们都不需要缺陷跟踪系统。这使他们可以专注于交付新功能，而不是维护系统。

还没有活文档

我采访**Cowans**的时候，他们系统的可执行需求说明的数量已经增长了许多，需要开始考虑重新组织需求说明，并在根本上形成活文档系统。**Cowans**说：

“刚开始建立的时候，我们没有充分考虑测试的高层结构。随着系统的增长，我们只是随意地按需增加新的测试。结果，在修改现有代码的时候，很难找到哪些测试覆盖哪些功能。决定网站功能的高层描述并依此组织测试，而不是简单地为每个新构建的功能添加新的测试，这会很有帮助。我认为这对开发产品与维护相对易于理解的代码都很有帮助，最终会形成描述新功能如何配舍现有功能的通用语言。”

17.2 当前的流程

Songkick的开发流程是基于看板的。他们的产品团队负责路线图，开发团队负责实现。产品团队由产品开发主管、1名创意总监和1名交互设计师组成。开发团队有9名开发人员和2名测试人员。开发团队中，2人偏重于客户端和用户界面，其他的偏重于中间件和后台。Cowans是CTO，也是开发团队的一员。据他说，公司试着在产品和开发之间融入尽可能多的协作，因此团队间的边界已经相当模糊了。

如果一个功能处在最高优先级，即将开始实现时，产品团队会在一起研究用户体验和实现它所需的技术。会议的产出就是线框图、特定用例的介绍，以及该功能用户故事列表的最初版本。

当开发团队有足够生产力开始实现功能的时候，他们会与产品团队及可能实现该功能的所有开发人员或测试人员一起进行首次会议。在这个会议上，他们把功能分解成用户故事，一起为每个故事讨论验收条件。验收条件被定义成一系列需要检查的点，包含之后需要完善的具体例子。

测试人员负责需求，包括用户故事和相关的验收条件列表。他们负责维护这些开发过程中的额外信息。由于可用性和用户交互比较重要，除运行所有的可执行需求说明外，他们会在开发后手工测试每个功能的核心功能点。因此在首次会议后测试人员就开始考虑测试计划了。

开发人员编写带实例的需求说明，测试人员进行审核，建议还有哪

些需要覆盖的。然后开发人员把它们都自动化，用TDD实现所需的功能，并让测试人员可以访问到这个代码分支。

然后测试人员运行手工测试，开始做探索测试，并向开发人员提供反馈。一旦测试人员和开发人员都认为功能已经准备好了，这个功能就进入等待集成的队列中。

队列中的功能会集成到主分支。然后运行整个持续验证，代码被部署到临时环境，测试人员为核心功能做最后的手工测试。之后，代码会上线进入生产环境。

顿悟之处

我询问**Cowans**，对他来说，实施实例化需求说明的关键啊哈时刻是什么时候。他说：

测试能看见的东西很容易，但是最终还需要对软件所做的事情有深刻的理解，而不仅仅是知道用户界面长什么样子。考虑使用用户故事和应用程序的路径确实有很大帮助。

将你的测试视为一等公民。它需要像应用程序代码本身一样认真维护。

测试是对应用程序所做事情的最可靠的描述。成功最终取决于是否构建了正确的东西并且做得很好。如果测试是对代码的描述，那么它们不仅仅是开发流程的重要部分，而且还是构建产品这个大流程的重要部分。他们能帮助你理解你已经构建了什么，并控制复杂度。

让流程中的所有人都参与进来很重要，这不仅仅是开发人员做的事。实例化需求说明最终提供了测试，这些测试是由开发人员编写的，并且产品负责人可以读得懂。你应该好好利用这一点。

17.3 重要的经验教训

对我而言，Songkick的重要经验是：如果没有臃肿的遗留系统拖累你，从TDD快速过度到实例化需求说明是有可能的。在Songkick，他们只是把它作为TDD流程覆盖到业务功能的扩展。

团队构建并维护网页系统，所以他们一开始自动化测试就是与用户界面紧密绑在一起的。这导致许多维护问题，也引导他们开始提炼需求说明，并在更高的抽象层次上自动化用户界面检查。

他们花了大概一年的时间才开始考虑建立活文档，并看到了当重写系统的一部分功能时，这一点是多么重要。

作为一家创业公司，Songkick专注于交付真正重要的事情，这让他们获得了不少好处。通过协作制定需求说明而获得的共识确保了他们全都专注于交付正确的产品。第二重要的好处是来自于可执行的需求说

明，因为这使他们能更早地发现问题，并能集中精力交付新功能，而不会浪费时间在修复缺陷上。

第18章 思想总结

我为编写这本书而开始的一系列调研，最初原因是为了在外界验证我的想法。我要记录下有许多团队因使用敏捷相关的技术而做出了很出色的产品。我所希望看到的是他们使用了BDD、敏捷验收测试，或者我所称为的实例化需求说明。我本以为自己已经对这些流程了如指掌，并且其他人的使用方式也都与我如出一辙。但是随着研究的不断深入，我逐渐改变了这种看法。为了让这些敏捷技术发挥作用，很多团队都对其做了调整以适应自己的使用环境。这也证明了并不存在所谓的“最佳实践”。软件开发与使用环境有很大的相关性，有些思想用在某些团队里得心应手，而用在其他团队里却水土不服。

回顾这些调研，关于如何高效地交付高质量的软件产品，我的收获多得令人惊讶。其中有些对我来说是全新的发现，而有些是用更宽广的视角来看待我所熟知的东西，这使得我对这些做法背后的实际推动力有了更深的理解。作为对本书的总结，我想提出我所学到的五大最重要方面。

18.1 协作制定需求能在项目干系人与交付团队之间建立信任

在Bridging the Communication Gap一书里，我提到需求说明工作坊有两种主要输出。一种是有形的：例子或者需求说明。另一种是无形的：通过交谈对需求达成的共识。我提到了这种无形的共识甚至可能比有形的例子还重要。结果却发现，真实情况要复杂得多，我在为本书做调研的过程中发现了另一种无形的输出。

uSwitch、Sabre、Beazley以及Weyerhaeuser的例子表明了协作制定需求在团队的氛围里引发了变化。结果，开发人员、分析师以及测试人员协作得更融洽，并且团队整合得比以往更好。引用Wes Williams的一句话，在协作制定需求之后“信任太棒了”。

我共事过的许多公司所使用的软件开发模型都是以互相不信任为前提的。商业用户将需求告诉分析师，却不信任分析师所做的详细说明，最后还需要对需求说明进行签收。分析师将需求告诉开发人员，却不信任开发人员交付的内容，因此测试人员需要使用比较独立的方式进行检查，以确定开发人员是否诚实。开发人员也不信任测试人员，测试人员报告缺陷后，开发人员不会立即检查代码，而只是说无法重现，或者

说“在我的机器上是好的”。所以测试人员被训练成不信任任何人，几乎就像个熟练的间谍一般。

一个基于互不信任的模型不仅会产生敌对的状态，还将滋生大量的官僚行为。据说，对需求进行签收是用户为了确保分析师所要做的事情是正确的。而事实上呢，签收只是为了防备以后出1现功能缺失时分析师受到责备！因为每个人都需要了解事情的进展情况，所以需求说明需要通过变更管理流程；而事实上，不会有人因为未将某项变更内容告诉他人而受到责备。据说，在测试过程中冻结代码是为测试人员提供一个更加稳定的环境。这也同样确保了开发人员不会由于在测试过程中修改了系统而受到责备。表面上，这整套体系是为了提供更好的质量。而事实上，这不3过是推诿的借口而已。

这纯粹是浪费！我们可以在商业用户、分析师、开发人员以及测试人员之间建立信任，这样可以消除推诿的借口以及随之而来的官僚作风。协作制定需求说明就是开始建立这种信任的一个上好的方式。

18.2 协作需要事先准备

在Bridging the Communication Gap一书中，除了提到举办“预计划会议”(pre-planning meeting)是实施迭代流程的一种好办法外，我并未更多地谈及工作坊的准备工作。我引入预计划会议是因为在每个工作坊开始之初，我们都会在确认例子的重要特性上花费大量的时间，而真正的讨论是在确立工作内容之后才正式开始的。而现在我知道了，除“预计划会议”外，还有其他更8多的做法。

有些团队在准备阶段使用了其他方式，在与他们交谈之后，我知道了“协作制定例子”的流9程有两个步骤。第一步，有人准备基础实例。第二步，团队讨论这些例子并进行扩展。准备阶段的目的是确保例子的基础问题已经解决，并且这些例子有固定格式方便团队进行阅读讨论。而这些事情可以只由一两个人完成，这样整个大的工作坊可以更加高效。

有些团队所在项目的需求模糊不清，需要大量的事先分析，因此他们的准备阶段会在协作工11作坊开始前2周开始。这样分析师可以与商业用户交谈，从他们那里收集实例，并开始提炼实例。而有些团队拥有更稳定的需求，他们只需提前几天准备实例，收集并解决一些明显的问题。这些方式都有助于更加高效地举办大型工作坊。

18.3 协作的方式多种多样

在Bridging the Communication Gap一书中，我推荐“大型的全体工作

坊”作为协作制定需求14说明的最佳方式。与工作在不同环境的团队交谈之后，我认识到了现实并非如此简单。

许多团队发现大型工作坊在一开始很有用，可以用于传递领域知识，还可以确保开发人员、测试人员、业务分析师以及项目干系人之间对预期结果有一致的认同。然而大多数团队在一段时间之后就不再举行大型工作坊了，因为他们发现这很耗时并且很难协调所有人的会议时间。

一旦整个体系成形、信任得到改善、开发人员与测试人员了解了很多领域知识之后，小型工作坊或者专门会议此时就足以产生好的需求说明了。许多团队的做法是“谁有兴趣谁参加”，会议人员只包含了接手该用户故事的相关人等。如果其他人需要变更某个功能，只需看活文档系统就能明白。

18.4 将最终目的视为业务流程文档，不失为一种有用的模型

如果我们将业务流程文档作为实例化需求说明的最终目的，那么许多常见的自动化问题与维护问题就不复存在了。例如，创建过于复杂的脚本来模拟软件的创建方式，这种做法的缺点就显而易见了。脚本最终都会变得难以维护，其用于沟通的价值微乎其微。

在社区里，我们几年前就发现了这个问题，并且许多实践者建议团队不要将验收测试编写成工作流。虽然对多数案例来说，此话不假，然而对于领域本身就是工作流的情况（如支付处理）却毫无帮助。David PeterSon看到了工作流在FIT里被滥用的情况，为此他专门编写了concordion以正视听，他的做法离“编写需求说明而非脚本”的观点又更近了一步。这是个不错的经验法则，但是对于开发网站来说却并不适用。问题在于它未能使验收测试（需求说明）模型与业务模型保持一致①，业务领域里的一个细微变动都会对测试产生霰弹效应，使得测试变得难以维护。

注释：①See http://dannorth.net/2O11/O1/31/whose-domain-is-it-anyway

如果我们专注于将业务流程文档化，需求说明的模型将与业务模型一致，并且变更也将对称。业务领域模型的一个细微变更只会在需求说明与测试里产生同样细微的变化。在开始编写软件之前，我们可以将业务流程很好地文档化，而即使所使用的技术改变了，需求说明也可维持不变。长期来看，阐述业务流程的需求说明更加物有所值。商业用户可以参与业务流程的文档化，并且相比与软件有关的验收测试，他们还可

以更好地提供反馈。

以上这些也告诉我们做什么自动化以及如何自动化。使需求说明中包含已创建的测试概念，或使其适应于用户界面交互，可以很容易地发现缺陷。如果需求说明记录了业务流程，那么自动化层就会在软件中运行这些业务流程。技术化流程、脚本以及模拟的界面交互要做的就是这些事情。自动化本身不是目的，它只是运行业务流程的一个工具而已。

为了创建可靠的文档，我们必须对其频繁验证。利用自动化可以做到频繁验证，并且不需要多少代价，但它不是唯一方式。有些东西无法很好地自动化，比如易用性。但是我们仍然可以对需求说明的一部分进行频繁验证，这样就解决了有些需求说明难以自动化的问题，这也是许多团队想要避免的问题。

18.5 活文档带来的长期价值

我采访的这些人几乎都经历过由更快交付与更高质量所带来的短期效益，但是“清理测试”的团队同样获得了相当不错的长期效益。作为一名咨询师，我帮助许多团队实施了这些实践，但是由于我没有与他们长期一起工作，所以我基本都无法体验到它们带来的长期效应。幸运的是，早期采用这些实践的团队到目前已经使用了有六七年的光景了，他们也已经看到了不错的长期效益。

爱荷华州助学贷款公司由于有了可靠的文档，所以可以很快地变更业务模型。ePlan Services团队在其中一名核心队员不在的情况下也能良好运作。Sierra项目的团队在接到支持请求时将“测试”用作支持文档。从这点来看，我认为不应该将他们使用的称作“测试”，因为不是用它们来1测试软件的，创建它们的目的是将它们用作可靠文档，并提供参考。

大多数团队在反复试验中采用了活文档系统，是由于他们要寻找更简单的方式来维护测试。2他们重组测试是为了测试更加稳定，使得测试与业务中的模型相一致。他们重组了包含测试的目录，这样在变更时能更方便地找到相关的测试，并且还用类似于商业用户考虑系统功能的组织方式演化了一个文档系统。

在这一点上，我可以很有信心地断言，如果一个新的团队特意从一开始就创建活文档系统，4而不是经过多年的试验与摸索后才去创建，那么他们很快就将大有斩获。

请铭记以上这点，我诚恳地邀请你与团队一起试试看。试过之后请与我分享你的体会。你可5以通过Email联系我，地址是gojko@gojko.com。

附录A 资源

图书

Gojko Adzic,Bridging the Communication Gap:Specification by Example and Agile Acceptance Testing(Neuri,2009).

Gojko Adzic,Test Driven.NET Development with FitNesse (Neuri,2008).

David Anderson,Kanban:Successful Evolutionary Change for Your Technology Business(Blue Hole Press,2010).

Mijo Balic,Ingrid Ottersten,and Peter Corrigan,Effect Managing IT(Copenhagen Business School Press,2007).

Mijo Cohn,Agile Estimating and Planning (Robert C.Martin Series) (Prentice Hall,2005)

Lisa Crispin and Janet gregory,Agile Testing:A Practical Guide for Testers and Agile Teams(Addison-Wesley Professional,2009).

Kev Darling,F-16 Fighting Falcon (Combat Legend) (The Crowood Press,2005).

Mark Denne and Jane Cleland-Huang,Software by Numbers:Low-Risk,High-Return Development (Prentice Hall,2003).

Eric Evans,Domain-Driven Design:Tackling Complexity in the Heart of Software (Addison-Wesley Professional,2003).

Steve Freeman and Nat Pryce,Growing Object-Oriented Software,Guided by Tests (Addison-Wesley Professional,2009).

Donald C.Gause and Gerald M.Weinberg,Exploring Requirements:Quality Before Design (Dorset House Publishing Company,1989).

Capers Jones,Estimating Software Costs:Bringing Realism to Estimating,2nd ed.(McGraw-Hill Osborne,2007).

Craing Larman and Bas Vodde,Practices for Scaling Lean & Agile Development:Large,Multisite,and Offshore Product Development with Large-Scale Scrum (Pearson Education,2010).

Rick Mugridge and Ward Cunningham,Fit for Developing Software:Framework for Integrated Tests (Prentice Hall,2005).

Mary Poppendieck and Tom Poppendieck,Lean Software Development:An Agile Toolkit (Addison-Wesley Professional,2003).

James Shore and Shane Warden,The Art of Agile Development (O'Reilly Media,2007).

Gerald Weinberg,Quality Software Management:Vol.1,Systems Thinking (Dorset House Publishing,1992).

在线资源

本书提供的所有在线资源的链接及更多可以在下面的网站上找到。

工具

Concordion:http//www.concordion.org.

Cucumber:http://cukes.info.

FitNesse:http://jbehave.org

Green Pepper:http://www.greenpeppersoftware.com.

JBehave:http://jbehave.org.

Robot Eramework:http://www.robotframework.org.

SpecFlow:http://www.specflow.org.

TextTest:http://www.texttest.org.

Twist:http://studios.thoughtworks.com/twist-agile-test-automation/.

视频

Gojko Adzic,"Challenging Requirements,"http://gojko.net/2009/12/10/challenging-requirements/.

Dan North,"How to Sell BDD to the Business,"http://skillsmatter.com/podcast/agile-testing/how-to-sell-bdd-to-the-business.

Hemal Kuntawala,"How we build quality software at USwitch.com,"http://skills matter.com./podcast/agile-testing-how-we-build-quality-software-at-uswitch-com.

Björn Regnell,"Supporting Roadmapping of Quality Requirements,"http://oredev.org/videos/supporting-roadmapping-of-quality-requirements.

演讲

Tim Andersen,"Persona Driven Development,"http://www.umsec.umn.edu/events/Code-Freeze-2010/PDD;http://timandersen.net/presentations/Persona_Driven_Developmen

Mark Durrand and Damon Morgen,"Creating a Lean Business from the inside out:Technical innovation at uSwitch.com.to reduce waste,"http://www.slideshare.nte/markdurrand/spa2010-uswitch.

文章

Gojko Adzic,"Agile in a Start-up Games Development

Studio,"http://gojko.net/2010/05/19/agile-in-a-start-up-games-development-studio/.

Gojko Adzic:Are tools necessary for acceptance testing,or are they just evil?http://gojko.net/2010/03/01/are-tools-necessary-for-acceptance-testing-or-are-they-just-evil.

Gojko Adzic,"Examples make it easy to spot inconsistencies,"http://gojko.net/2009/05/12/examples-make-it-easy-to-spot-inconsistencies/.

Gijko Adzic:How to implement UI testing without shooting yourself in the foot,http://gojko.net/2010/04/13/how-to-implement-ui-testing-without-shooting-yourself-in-the-foot-2/.

Gijko Adzic:Improving testing practices at Google,http://gojko.net/2009/12/07/improving-testing-practices-at-google/.

Gojko Adzic,"QUPER model for better requirements,"http://gojko.net/2009/11/04/quper-model-for-better-requirements/.

Gojko Adzic,"Shock therapy agile agile adoption at 7Digital,"http://gojko.net /2009/12/08/shock-eherapy-agile-adoption-at-7digital/.

Michael Bolton,"Acceptance Tests:Let's Change the Title,Too,"http://www.developsense.com/blog/2010/08/acceptance-tests lets-change-the-title-too/.

Michael Bolton,"Testing vs.Checking,"http://www.developsense.com/blog/2009/08/testing-vs-checking/.

Alistair Cockburn,"Sacrifice One Person,"http://alistair.cockburn.us/Sacrifice+one+person+strategy.

Craig Larman and Bas Vldde,"Acceptance Test-Driven Development with Robot Framework,"http"//code.google.com/p/robotframework/wiki/ATDDWith RobotFrameworkArticle.

Craig Larman and Bas Vodde,"Feature Teams Primer,"http://www.featureteams.org/feature_team_primer.pdf.

Dan North,"Whose domain is it anyway?"http://dannorth.net/2011/01/31/whose-domain-is-it-anyway/.

Björn Regnell,Richard Berntsson Svensson,and Thomas Olsson,"Supporting Roadmapping of Quality Requirements,"IEEE Software

25,no.2(Mar/Apr 2008):43-47

James Shore,"Alternatives to Acceptance Testing,"http://jamesshore.com./Blog/Alternatives-to-Acceptance-testing.html.

James Shore,"The Problems with Acceptance Testing,"http://jamesshore.com/Blog/The-Problems-With-Acceptance-Testing.html.

Lance Walton,"Writing Maintainable Acceptance Tests,"http://www.casual micacles.com/blog/2010/03/04/writing-maintainable-acceptance-tests/.

漫画

Chris Matts,"Real Options at Agile 2009,"http"//www/lulu.com/product/file-download/real-options-at-agile-2009/5949486.

培训课程

Gojko Adzic:http://neuri.co.uk/training.

Object Mentor:http://objectmentor.com/omTrining/omi_training_index.html.

Lisa Crispin and Janet Grgory:http://www.janetgregory.ca/training.htm.

Elisabeth Hendrickson:http://www/qualitytree.com/workshops/.

Pyxis Technologies:http://pyxis-tech.com/en/our-offer/training.

Tech Talk:http://www/techtalk.at/training.aspx.

Rick Mugridge:http://www.rimuresearch.com/Coaching.html.

Table of Contents